Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Robótica educativa y pensamiento computacional

Educational robotics and computational thinking

Cristian Manuel Ángel Díaz

La Laguna, 3 de julio de 2019 Grado en Ingeniería Informática

D. **Eduardo Manuel Segredo González**, con N.I.F. 78.564.242-Z profesor Ayudante Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

D. **Rafael Arnay del Arco**, con N.I.F. 78.569.591-G profesor Ayudante Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor.

C E R T I F I C A N

Que la presente memoria titulada:

"Robótica educativa y pensamiento computacional"

ha sido realizada bajo su dirección por D. **Cristian Manuel Ángel Díaz**, con N.I.F. 43.484.608 -A.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 3 de julio de 2019.

Agradecimientos

En primer lugar, quiero dar las gracias a mis padres, por haberme permitido con mucho esfuerzo hacerme posible estudiar este grado, y en general a mi familia por estar siempre a mi lado.

En segundo lugar, cada uno de mis compañeros y amigos de la carrera, por tenderme la mano siempre que lo he necesitado, tanto en lo personal como en lo académico, muchas gracias por estos años, ha sido toda una experiencia.

Por último, quisiera agradecer a mis tutores Rafael y Eduardo por proponer este TFG, además de orientarme y ayudarme durante el desarrollo de este.

Licencia

Resumen

El objetivo de este Trabajo de Fin de Grado (TFG) ha sido la creación de una herramienta web libre y gratuita para facilitar a cualquier centro educativo la enseñanza de conceptos básicos sobre robótica y programación dentro de sus planes de estudios. Dicha herramienta permite diseñar y personalizar un robot añadiendo sensores a su carrocería. Tras su creación, se podrá poner a prueba en un entorno de simulación con distintos retos. En dicho entorno podremos definir el comportamiento del robot por medio de programación visual basada en bloques, en los que podremos juntar las instrucciones y la información recogida por los sensores que hayamos añadido a nuestro diseño para superar los desafíos propuestos.

Este proyecto sigue una línea de trabajo ya establecida en el anterior curso académico, ya que la herramienta producto de este TFG será la unificación de tres trabajos, los cuales solían ser módulos independientes (creación, simulación y definición de comportamiento), que ahora mismo se encuentran completamente integrados dentro de una misma aplicación.

Palabras clave: Robótica Educativa, Pensamiento Computacional, Simulador, Programación Visual, Unity, C#, Blockly, UBlockly.

Abstract

The goal of this final degree project has been the development of a free web tool to facilitate to any school teaching basics about robotics and programming within its curriculum. This tool allows designing and customizing a robot adding sensors to the body. After its creation, it can be tested in a simulation environment with different challenges. In this environment, we will define the behaviour of the robot by means of a block based visual programming language, in which we will be able to gather the instructions and the information gathered by the sensors that we have added to our design to overcome the challenges proposed.

This project follows a line of work already established in the previous academic year, since the tool product of this final degree project will be the unification of three works, which used to be independent modules (creation, simulation and definition of behaviour), that are now fully integrated into the same application.

Keywords: Educational Robotics, Computational Thinking, Simulator, Visual programming, Unity, C#, Blockly, UBlockly.

Índice general

Capítulo 1 Introducción 13
1.1 Motivación 13
1.2 Objetivos 13
1.3 Esquema de módulos de la aplicación 14
1.3.1 Diseño del robot 14
1.3.2 Programación visual del robot 15
1.3.3 Simulador del robot 15
Capítulo 2 Contexto 16
2.1 Conceptos de interés 16
2.1.1 Robótica Educativa 16
2.1.2 Pensamiento Computacional 17
2.2 Proyectos relacionados 17
2.2.1 V-Rep 17
2.2.2 CoderZ 18
2.2.3 Dash 18
2.3 Contribuciones 19
Capítulo 3 Metodología 20
3.1 Fases de desarrollo 20
3.2 Tecnologías y recursos utilizados 21
3.2.1 Unity 21
3.2.2 UBlockly 22
3.2.3 GitHub 25
3.2.4 Módulos de la aplicación 25
3.3 Desarrollo del proyecto 25
3.3.1 Unión de los módulos de Creación y Simulación 27
3.3.2 Unión de los módulos de Unity con el módulo de programación visual 29
3.3.3 Mejoras de funcionalidades e interfaz 37
Vista de las etiquetas de sensores 37
Creación de esquema del robot en entorno de simulación 38
Cambio de tamaño de la ventana del escenario de simulación 39
Reiniciar escena 40
Vuelta a la escena de creación 41
Ocultar bloques de sensores no presentes en el diseño del robot 41

Aumentar y disminuir el tamaño de los bloques de área de trabajo 42
3.3.4 Creación de retos 43
Reto del sensor infrarrojo 43
Reto del sensor de contacto y sensor ultrasonido 44
3.3.5 Exportar proyecto a HTML5 46
Capítulo 4 Caso de uso 47
Capítulo 5 Conclusiones 50
Capítulo 6 Summary and Conclusions 51
Capítulo 7 Presupuesto 52
7.1 Presupuesto 52

Índice de figuras

Figura 1: Esquema de relación entre los tres módulos 14
Figura 2: Módulo de diseño 15
Figura 3: Módulo de programación visual 15
Figura 4: Módulo de simulación 15
Figura 5: Vrep 18
Figura 6: CoderZ 18
Figura 7: Dash 18
Figura 8: Esquema de funcionamiento de UBlockly 23
Figura 9: Fichero de definición de bloque 24
Figura 10: Fichero de diccionario 24
Figura 11: Ejemplo de bloque 24
Figura 12: Esquema de funcionalidad y relación de los módulos actuales 26
Figura 13: Permanente en la escena de creación 27
Figura 14: Permanente en la escena de simulación 27
Figura 15: Botón de comienzo de simulación 28
Figura 16: Sensor láser colisionando por la parte trasera con el modelo 3D del robot 28
Figura 17: Estructura de directorios conjunta de los módulos de simulación y creación 29
Figura 18: Estructura de directorios con UBlockly integrado 30
Figura 19: Cambio de tamaño de la ventana del entorno de simulación 30
Figura 20: Escena de simulación con UBlockly integrado 31
Figura 21: Estructura del Workspace 31
Figura 22: Fichero definición de toolbox (toolbox_robot) 31
Figura 23: Fichero definición de bloque (moveBlocks) 31
Figura 24: Fichero diccionario (move_en) 31
Figura 25: BlockResSettings diccionario 32
Figura 26: BlockResSettings bloques 32
Figura 27: BlockResSettings toolbox 32
Figura 28: Construir bloques 32
Figura 29: Bloque de movimiento 32
Figura 30: Fichero de asignación de función (RobotCmds) 33
Figura 31: Fichero de definición de comportamiento (RobotController) 33
Figura 32: Redefinición MoveForward (RobotController) 34
Figura 33: Condiciones de parada de giro a la izquierda (RobotController) 34

Figura 34: Código de detención de movimiento (RobotController) 35
Figura 35: Repertorio de bloques de la categoría MOVE 35
Figura 36: Renombrar sensores instanciados para su etiquetado 36
Figura 37: Función de asignación sensor ultrasonido (RobotCmds) 36
Figura 38: Función de asignación sensor ultrasonido (RobotController) 36
Figura 39: Repertorio de bloques de la categoría SENSOR 37
Figura 40: Escena de simulación con las categorías MOVE y SENSOR 37
Figura 41: Añadiendo etiqueta de sensor 37
Figura 42: Modificando texto de la etiqueta 37
Figura 43: Cambio de color de las etiquetas de los sensores 38
Figura 44: Desactivación de etiquetas (RobotCollider) 38
Figura 45: Etiquetas de sensores en el robot 38
Figura 46: Creación y configuración de scripts de Robot_Esquema (RobotScriptController) ..39
Figura 47: Cambio de posición, rotación y restricciones de Robot_Esquema (Rotation) 39
Figura 48: Esquema del robot en la escena de simulación 39
Figura 49: Cambio de tamaño de la ventana del entorno de simulación 40
Figura 50: Botón de cambio de tamaño de ventana 40
Figura 51: Función de reinicio de la simulación 40
Figura 52: Botón de reinicio de simulación 40
Figura 53: Regreso a la escena de creación y parada del intérprete 41
Figura 54: Botón de retorno a la escena de creación 41
Figura 55: Fragmento de código de asignación de lados del robot al fichero CrearSensor.cs ..41
Figura 56: Modificación para ocultar bloques de sensores (ClassicToolbox.cs) 42
Figura 57: Categoría SENSOR con los bloques relacionados con los sensores del robot..... 42
Figura 58: Cambiar escala del panel de codificación 42
Figura 59: Botones para cambiar escala del panel de codificación 42
Figura 60: Entorno de simulación: Reto para Sensor Infrarrojo 43
Figura 61: Plane negro superpuesto 43
Figura 62: Configuración de robot: Reto para Sensor Infrarrojo 44
Figura 63: Script para superar prueba: Reto para Sensor Infrarrojo 44
Figura 64: Entorno de simulación (Vista aérea): Reto para Sensor de Contacto y Ultrasonido 45
Figura 65: Entorno de simulación (Vista robot): Reto para Sensor de Contacto y Ultrasonido 45
Figura 66: Configuración de robot: Reto para Sensor Ultrasonido 45
Figura 67: Configuración de robot: Reto para Sensor de contacto 45
Figura 68: Script para superar prueba: Reto para Sensor de contacto..... 45
Figura 69: Script para superar prueba: Reto para Sensor Ultrasonido 45
Figura 70: File->Build Settings en el editor de Unity 46
Figura 71: Ventana de exportación de proyecto 46
Figura 72: Proyecto en el navegador 46

Figura 73: Menú de inicio 47
Figura 74: Añadiendo un sensor de ultrasonido a la parte delantera del robot 47
Figura 75: Personalizando sensor ultrasonido 48
Figura 76: Preparativos para iniciar el reto de sensor ultrasonido 48
Figura 77: Llevar un bloque al área de trabajo 48
Figura 78: Encajar dos bloques 49
Figura 79: Cuadro de diálogo: Introducción de número 49
Figura 80: Cuadro de diálogo: Selección de una opción 49
Figura 81: Ejecutando bloques del área de trabajo 49

Índice de tablas

Tabla 1: Diagrama de Gantt de las actividades del proyecto 20
Tabla 2: Tabla de presupuesto 53

Capítulo 1
Introducción

1.1 Motivación

Saber programar hoy en día es una necesidad, ya que todos estamos vinculados con la tecnología, sea cual sea nuestra ocupación. Estar formados en esta disciplina permitirá que nos podamos relacionar de forma más natural con el mundo que nos rodea, además de mejorar la creatividad, la capacidad de pensamiento crítico y estructurado y la habilidad de resolver problemas a través del **pensamiento computacional** [3][4], sin olvidarnos de que nos puede abrir muchas puertas en el mundo profesional.

Por estos motivos y muchos otros, la enseñanza de este tipo de competencias es más necesaria que nunca, por lo que debemos introducirlas desde edades tempranas para que las generaciones venideras estén preparadas para la sociedad de las nuevas tecnologías.

La motivación principal del desarrollo de este TFG surge de la posibilidad de contribuir a que impartan asignaturas de programación en los currículos educativos de enseñanza obligatoria. La aportación que se haría por nuestra parte sería ofrecer un software de simulación libre y gratuito que tan solo necesite de un ordenador con conexión a internet.

En nuestro caso, hemos apostado por la **Robótica Educativa** [1][2][6] para la enseñanza de las competencias anteriormente mencionadas. Las clases de este tipo de materias comienzan con la propuesta de un reto que debe ser superado por el alumnado por medio de su ingenio y las herramientas con las que cuenten para hacer que el robot supere dicha prueba.

Con la herramienta producto de este TFG se podrá crear un robot que podremos personalizar añadiendo sensores, por medio de los cuales podremos interactuar con el medio y recoger datos, que nos serán útiles para poder superar los retos propuestos.

La definición del comportamiento que seguirá el robot para superar las pruebas será a través de programación visual basada en bloques. Tendremos la posibilidad de usar bloques para definir desde elementos básicos de la programación, como son los bucles o los condicionales, hasta el propio repertorio de acciones que puede ejecutar el robot, que serían avanzar, retroceder, girar y detenerse. La combinación de los diferentes bloques disponibles permitirá el diseño de diferentes programas cuyo objetivo será superar el reto planteado.

1.2 Objetivos

El objetivo de este TFG consiste en la unificación e integración de tres trabajos desarrollados el pasado curso académico en el ámbito del grado de Ingeniería Informática de la Universidad de La Laguna, además de conseguir que se pudiera llevar dicho proyecto a navegadores.

Los tres trabajos en cuestión serían:

- *"Framework para la creación de un robot modular"*, llevado a cabo por **Sergio García de la Iglesia.**
- *"Simulador de robots para fomentar el pensamiento computacional a través de lenguajes de programación visual"*, desarrollado por **Eduardo de la Paz González [27].**

- "*Robótica Educativa mediante Programación Visual",* elaborado por **Andrea Rodríguez Rivarés [26].**

Cada uno de estos trabajos constituye un módulo con una funcionalidad específica, ya que podríamos decir que, a pesar de no estar unidos y tener que comunicarse a través de un sistema de paso de ficheros, formaban parte de una misma aplicación (en apartados posteriores se detallan más aspectos acerca de su funcionalidad).

Y relacionado con este objetivo, tenemos otro, que es proporcionar una aplicación gratuita y fácil de incorporar a los centros educativos. Por este motivo, nos hemos decantado por implementarla como aplicación web, y de este modo, permitir que llegue a la mayor cantidad de usuarios posible, sin la necesidad de involucrar piezas físicas de robot para poder crearlos.

Todo esto con el afán de ayudar a que se puedan desarrollar competencias relacionadas con el pensamiento computacional, ya que las barreras económicas desaparecerían al solo requerir de un software sin coste y de los ordenadores con los que cuente el centro. La aparición de proyectos de esta temática puede motivar a otros desarrolladores a crear herramientas que contribuyan a este fin.

1.3 Esquema de módulos de la aplicación

A continuación, hablaremos en detalle de las funcionalidades específicas y de cómo se relacionaban entre sí los módulos de la aplicación justo antes de comenzar el desarrollo de este TFG (**Figura 1**).

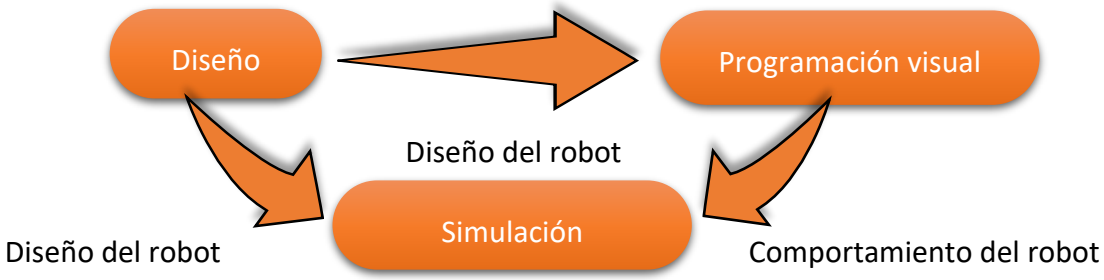


Figura 1: Esquema de relación entre los tres módulos

1.3.1 Diseño del robot

Este módulo permite la creación y diseño de un robot con chasis rectangular, que consta de tres ruedas (dos en la parte anterior y una en la parte posterior). Dicho robot podrá ser modificado a antojo del usuario, como por ejemplo editando el radio de las ruedas, o incluso realizando cambios en las dimensiones del chasis (alto, largo y ancho).

Además de lo mencionado, también tendremos la posibilidad de añadir sensores al robot (**láser, ultrasonido, infrarrojo, de contacto y telémetro láser**), que podrán ser localizados en cualquier parte de este. También podremos alterar los parámetros de estos sensores, modificando su radio de acción o la precisión que tienen al tomar datos del medio.

Tras terminar nuestro diseño, existe un botón de exportación, con el que se generará un fichero XML con la información del robot. Dicho fichero será cargado posteriormente de forma manual por los siguientes módulos de la aplicación (**Figura 2**).

Figura 2: Módulo de diseño

1.3.2 Programación visual del robot

Este segundo módulo permite programar el comportamiento del robot usando programación visual. En concreto, se ha usado la librería de Google Blockly [5] para diseñar un lenguaje de programación específico para este propósito.

Estos bloques contienen tanto las acciones que puede realizar el robot (avanzar, retroceder, girar) como referencias a los sensores que hayamos añadido a nuestro diseño. Gracias a los sensores podremos condicionar las acciones que realizará el autómata en función de la información que devuelvan del medio.

Una vez se haya finalizado la creación de nuestro código, podremos generar un fichero JavaScript, que contendrá el código fuente del programa desarrollado, con el fin de poder cargarlo más adelante en el módulo de simulación (**Figura 3**).

Figura 3: Módulo de programación visual

1.3.3 Simulador del robot

En el simulador cargaremos los ficheros creados en los módulos anteriores, permitiendo que se genere una copia del robot que hemos diseñado y se asigne el conjunto de instrucciones que este seguirá. Tras este paso, debemos iniciar la simulación, en la que podremos observar cómo se desenvuelve el robot en el entorno con ayuda de sus sensores.

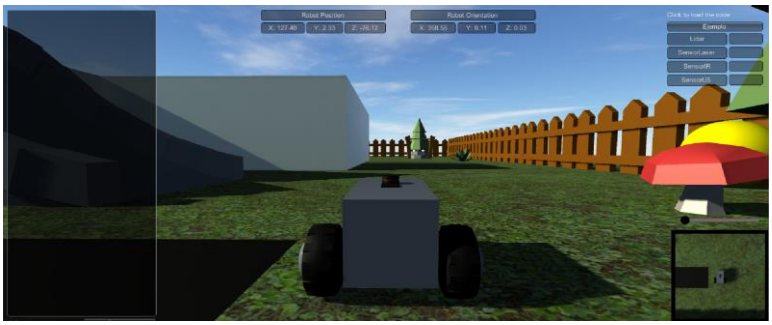


Figura 4: Módulo de simulación

Capítulo 2
Contexto

2.1 Conceptos de interés

En esta sección hablaremos acerca de dos temas fundamentales para la rama de conocimiento en la que catalogamos este TFG, como son la Robótica Educativa y el Pensamiento Computacional. Definiremos ambos conceptos para comprender un poco mejor como se relacionaentre sí.

2.1.1 Robótica Educativa

La Robótica Educativa es una metodología didáctica que usa a los robots como medio para el impulso de capacidades y competencias a través de la resolución de retos. Dichos retos deberán ser superados por el alumnado haciendo acopio del ingenio y las herramientas que les sean proporcionadas durante esta actividad.

Por lo general, en las clases de Robótica, el docente comienza dando unas nociones básicas a los estudiantes, para a continuación proponer un problema que debe ser superado. El aula en el que se esté desarrollando la actividad contará con todo el material didáctico necesario para hacer más sencillo el desarrollo de esta (además del material para poder desarrollar el robot), como por ejemplo vídeos explicativos, instrucciones o consejos acerca de cómo afrontar el problema.

Podemos separar en distintas fases el desarrollo de una clase de robótica educativa:

1. **Afrontar el problema:** Se trata de investigar y analizar el reto al que nos enfrentamos para comenzar a buscar una solución.
2. **Diseño y creación del robot:** A raíz de lo descubierto en la primera fase de análisis del problema, comenzaremos a diseñar y crear un robot que reúna los requisitos necesarios para superar el reto planteado, teniendo en mente cuál debe ser el comportamiento de este y si tiene los componentes necesarios para poder ejecutar las acciones que le pediremos en el siguiente punto.
3. **Programación de órdenes:** Una vez terminada la fase de creación y diseño, debemos programar un pequeño script que permita al robot comportarse y moverse, de modo que consiga resolver el problema en un entorno de simulación.
4. **Prueba de simulación:** En este punto se deberá probar tanto el diseño como el comportamiento programado, y si fuera necesario, se deben hacer cambios en ambos hasta conseguir un resultado satisfactorio. Si el robot supera el reto, querrá decir que hemos encontrado una de las múltiples soluciones a la prueba.
5. **Documentar y presentar:** Se muestran evidencias de que el robot ha superado la prueba de simulación y el alumnado comparte el diseño a la clase como alternativa para superar el problema.

Examinando las distintas fases que tiene una clase de robótica, podemos observar que se trabajan una gran cantidad de habilidades. Por un lado, las más técnicas, como la capacidad de resolución de problemas usando el pensamiento computacional, la lógica y el razonamiento. Por otra parte, se fomentan habilidades sociales y comunicativas, entre ellas el trabajo en equipo y el habla en público.

A modo de síntesis, podríamos decir que es un sistema de enseñanza que no es convencional, ya que tiene un componente lúdico, que en este caso serían los retos presentados a modo de juego. Este añadido la convierte en una metodología ideal a la hora de introducir al alumnado a la robótica, y en general, a la tecnología.

2.1.2 Pensamiento Computacional

Podemos definir el Pensamiento Computacional como la capacidad de un individuo de afrontar un problema por medio del uso de habilidades relacionadas con las Ciencias de la Computación, como el pensamiento algorítmico, la abstracción, la descomposición y el reconocimiento de patrones entre otras.

Tal y como dijo Beatriz Ortega-Ruipérez: "*La persona que emplea un pensamiento computacional puede descomponer el problema en pequeños problemas que sean más fáciles de resolver, y reformular cada uno de estos problemas para facilitar su solución por medio de estrategias de resolución de problemas familiares*." [7].

Podemos sacar como conclusión que usando este método. podremos resolver problemas complejos de forma más efectiva a la habitual.

A continuación, se definirán los pasos usados para la resolución de problemas mediante el Pensamiento Computacional:

- **Descomposición del problema**: Consiste en dividir el problema original en problemas que sean fácilmente solucionables. La solución conjunta de todos estos pequeños problemas conseguirá resolver el problema original.
- **Reconocimiento de patrones:** El siguiente paso es tratar de reconocer similitudes entre los distintos pequeños problemas, que servirán para atajar su resolución en el caso de que otros compartiesen la misma solución o parte de ella.
- **Abstracción:** Se trata del grado de detalle con el que tratamos el problema analizando sus propiedades por capas ignorando el resto, de modo que podamos destacar sus propiedades específicas, diferenciándolas del conjunto. [8]
- **Pensamiento algorítmico:** A raíz de las fases anteriores, se plantearán una serie de pasos (algoritmo) que logren resolver el problema procurando generalizar. Esto permitirá que problemas similares puedan ser resueltos con un algoritmo igual o semejante al planteado en un primer momento.

2.2 Proyectos relacionados

Este TFG persigue la creación de una herramienta de software que permita diseñar, programar y simular un robot para servir como introducción a la robótica y a la programación. No obstante, existen otros proyectos que tienen un propósito semejante al nuestro. En los siguientes puntos veremos algunos de los más importantes.

2.2.1 V-Rep

Es un simulador de robótica cuya interfaz puede recordar a la de Unity. Para el scripting podemos usar varios lenguajes bastante conocidos, entre los que se encuentran C/C++, Python, Java o Matlab.

Nos brinda la posibilidad de implementar de forma rápida nuestros diseños de autómatas y permite un uso tanto a nivel industrial como a nivel educacional. Cuenta con muchas funcionalidades, como cálculos de cinemática directa/inversa o la capacidad de agregar sensores a

nuestro robot para poder interactuar con el entorno. Es compatible con Linux, MacOS y Windows y cuenta con una versión de código abierto y otra con opciones más avanzadas[11][12] (**Figura 5**).

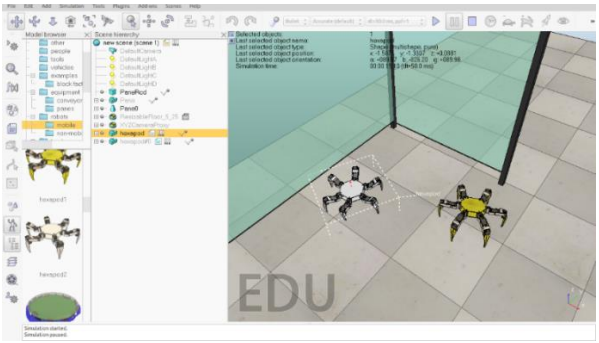


Figura 5: Vrep

2.2.2 CoderZ

Es una solución basada en la nube orientada exclusivamente al ámbito educacional. Permite usar tanto Blockly para los iniciados como la codificación en Java para la creación de robots virtuales en 3D.

Cuenta con un plan de estudios definido y muchas herramientas que ayudan a la enseñanza a través de este entorno, como presentaciones, modelos de evaluación y plantillas que permiten hacer demostraciones. No es una solución gratuita, lo que puede dificultar a aquellos centros educativos más modestos tener acceso a esta herramienta [9][10] (**Figura 6**).

Figura 6: CoderZ

2.2.3 Dash

A priori podría parecer un simple juguete para niños, pero este producto está lleno de posibilidades. Cuenta con una gran cantidad de sensores con los que podrá moverse y sortear obstáculos. Al estar montado y permitir su programación a través de dispositivos móviles con iOS o Android por medio de Blockly, lo hace accesible para los niños y permite que los padres aporten su grano de arena al ser muy fácil de usar. Es una buena solución para el hogar, a pesar de que el precio sea algo elevado [13] (**Figura 7**).

Figura 7: Dash

2.3 Contribuciones

En este apartado nos centraremos en enumerar las distintas aportaciones que brinda este TFG respecto al resto de recursos disponibles actualmente.

Gratuito

Resulta muy complicado instaurar soluciones educativas de este tipo en los centros estudiantiles, ya que la mayoría no disponen de los fondos necesarios para contar con estos recursos.

Este hecho provoca que solo unos pocos puedan disponer de estas herramientas. La situación ante la que nos encontramos resulta bastante injusta, ya que todo el alumnado debería tener derecho a recibir la misma educación, independientemente del presupuesto del colegio y la situación económica de las familias de los estudiantes.

Hacer gratuito el material para impartir asignaturas relacionadas con el ámbito de las nuevas tecnologías permitirá que todos los centros educativos puedan adoptarlas en su plan de estudios. Con esto se podrá lograr una formación igualitaria y competente para los nuevos tiempos.

Software libre

La gran mayoría de proyectos de software educacional han sido desarrollados con fines comerciales, por lo que el código o cualquier recurso de la herramienta suele ser privado.

Este proyecto ha sido creado sin afán de lucro, por lo que tendrá licencia de código abierto, permitiendo que otros desarrolladores que estén interesados puedan aprovechar, modificar o mejorar esta herramienta. Con este movimiento se pretende conseguir uno de los objetivos de este TFG, que es aumentar la cantidad de proyectos gratuitos relacionados con la robótica educativa.

Simplicidad

Por lo general, este tipo de simuladores suelen ser bastante complejos, tanto por las funcionalidades que ofrecen como por la interfaz del software, provocando una barrera para los más inexpertos. V-Rep es un ejemplo típico de este tipo de simuladores.

Nosotros ofrecemos una solución informática que sea accesible para todo tipo de públicos, ya que, en definitiva, es una simplificación de los simuladores convencionales.

Capítulo 3
Metodología

3.1 Fases de desarrollo

La distribución de tareas de la fase de desarrollo fue planteada de la siguiente forma desde la elaboración del anteproyecto de este TFG y siguiendo la cronología de la **Tabla 1**:

- **Actividad 1:** Buscar la bibliografía necesaria para obtener conocimiento acerca de la Robótica Educacional y la relevancia que puede tener en el sistema educativo, tanto a nivel de oportunidades laborales como en el desarrollo de habilidades relacionadas con el pensamiento computacional. A su vez, realizar una búsqueda acerca de trabajos que tengan un propósito similar a la herramienta que se pretende desarrollar en este proyecto, lo que ayudará a la hora de plantearnos el diseño del programa o crear mejoras a los ya existentes.
- **Actividad 2:** Estudiar las tecnologías necesarias y familiarizarse con los proyectos a unificar. Antes de comenzar el desarrollo, es necesario conocer Unity y Blockly, ya que fueron los pilares en la creación de los módulos que se tratan de integrar. Además, se deberá investigar cómo funcionan los mismos para poder unirlos correctamente.
- **Actividad 3:** Integración completa de los módulos de simulación y creación del robot en Unity. Se pretende unir estos en un único módulo con el fin de hacer más sencilla la labor de exportarlos a código HTML, además de hacer que se comuniquen mejor. A medida que se vayan integrando ambos módulos se deben ir solucionando las posibles incompatibilidades que puedan tener con los navegadores. Se perderá el intérprete de JavaScript con el que cuenta el simulador, ya que no será necesario.
- **Actividad 4:** Desarrollo e integración del módulo unificado de Unity con el módulo de Blockly en el navegador. Al haber eliminado el intérprete de JavaScript del simulador de robots, se pretende que éste acepte órdenes directas del módulo de Blockly en tiempo de ejecución. Se deberá encontrar el intérprete que permita llevar a cabo dicho diálogo, haciendo que sea compatible con los navegadores. Una vez hecho esto, se creará una *landing page* que presentará el proyecto listo para ser usado.
- **Actividad 5:** Elaboración de la memoria de trabajo, que se deberá ir rellenando conforme el proyecto vaya evolucionando con la bibliografía de las investigaciones, así como información relativa al desarrollo del proyecto e información de uso.

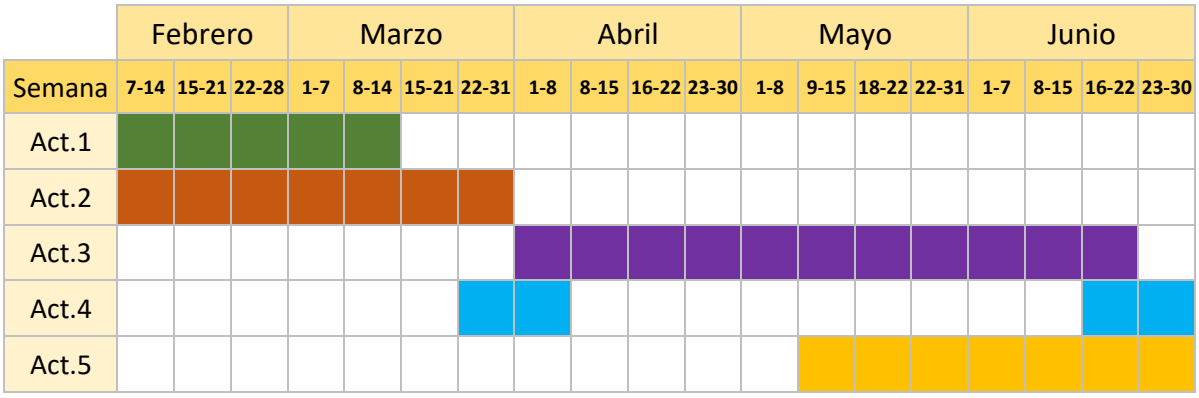


Tabla 1: Diagrama de Gantt de las actividades del proyecto

3.2 Tecnologías y recursos utilizados

En los siguientes apartados nombraremos y explicaremos las tecnologías más importantes utilizadas para la realización de este TFG, desde los recursos usados para la creación de la herramienta, hasta las tecnologías de apoyo al desarrollo del proyecto.

3.2.1 Unity

Unity [14][15][16] es un motor gráfico desarrollado por Unity Technologies. Es uno de los más usados a nivel mundial para la creación de contenido interactivo tanto 2D como 3D, generalmente videojuegos. Está disponible para la gran mayoría de sistemas operativos (Windows, MacOS y Linux) y los proyectos pueden ser exportados a una gran lista de plataformas. Entre las más destacadas podemos nombrar los dispositivos móviles, las videoconsolas de última generación o los principales navegadores web.

Para este proyecto estamos usando la versión Unity Personal 2019.1.0f2 (64-bit) de Unity3D, ya que para la simulación del robot requeríamos de un entorno tridimensional en el que pudiéramos representar las posibles colisiones que pudiera tener el autómata con los elementos del entorno. Con Unity3D la codificación se puede hacer a través de los lenguajes C# y JavaScript, que van de la mano con el *IDE* (Entorno de desarrollo integrado) *MonoDevelop*. Además, cuenta con un editor visual muy potente e intuitivo que nos ayudará a agilizar el desarrollo de nuestros proyectos, pudiendo crear con facilidad nuestros propios entornos 3D, o importar recursos creados por otros desarrolladores de la comunidad a través de la *Asset Store*. Estos recursos van desde modelos 3D, hasta texturas o efectos de sonido.

Hasta el comienzo del desarrollo de este TFG no había tenido contacto con Unity ni con el lenguaje C#, usado para la codificación de los scripts del proyecto, aunque gracias a la curva de aprendizaje de esta tecnología y a la gran cantidad de recursos online disponibles, la adaptación a esta tecnología fue más llevadera.

Vamos a pasar a definir algunos de los conceptos más importantes de Unity para entender un poco mejor cómo funciona este motor gráfico:

- **GameObject:** Es el elemento principal de Unity. Dichos elementos se encuentran contenidos dentro de las escenas. Como tal, el GameObject es un elemento vacío, que de forma nativa solo cuenta con un componente que define su posición, rotación y escala. Debemos añadirle otros componentes para cambiar las propiedades y el comportamiento de este en la escena. Un GameObject podría ser desde una piedra o un árbol, hasta una luz que ilumine los elementos de la escena.
- **Componente:** Son aquellos elementos que al añadirse a un GameObject modifican sus propiedades. Estos son los algunos de los componentes más importantes.
    - **Scripts:** Por medio de la codificación de scripts podremos tanto alterar parámetros de nuestro objeto como modificar el comportamiento de este en la escena, como por ejemplo haciendo que responda con una determinada acción tras la pulsación de una tecla [17].
    - **Transform:** Es un componente obligatorio para cualquier GameObject. Sirve para localizarlo dentro de la escena, además de para cambiar su rotación y su escala.
    - **Mesh:** Este elemento permite definir la malla de puntos con la que podremos establecer cómo será renderizado el GameObject en la escena.
    - **Collider:** Con este componente podremos configurar una malla de colisión para un GameObject. Si dos mallas de este tipo se encuentran, podemos decir que ha habido

contacto entre ellas. Cabe destacar que la malla de colisiones es totalmente independiente a la malla de renderizado de un objeto.
    - **Rigidbody:** Al añadir un componente de este tipo a un objeto, estaremos indicando que se encuentra bajo el control de las físicas del motor 3D. Esto quiere decir que los GameObjects que tengan dicho componente se verán influenciados por la gravedad y otro tipo de fuerzas, que pueden ser tanto fruto de una colisión con un elemento de la escena o por medio de su aplicación vía script [18].
- **Escena:** Contiene los entornos y menús creados a base de GameObjects. Cada escena debe contener al menos una cámara para poder visualizar lo que esté pasando durante la ejecución del programa y una luz direccional para iluminar el entorno [20].
- **Prefab:** Tras haber creado y modificado un GameObject tenemos la posibilidad de guardar un modelo de este con todas sus propiedades. Este modelo puede ser usado posteriormente para ser instanciado en las escenas de nuestro proyecto. Además, si modificamos un *Prefab*, automáticamente todas las instancias que existan de este tendrán los cambios que hemos añadido a nuestras escenas [19].

3.2.2 UBlockly

UBlockly [21] es una reinterpretación de Google Blockly desarrollada con Unity. Es un proyecto creado por el desarrollador Ling Mao que permite llevar de una forma más sencilla y eficiente la programación por bloques de Blockly a Unity.

En un inicio se barajó la posibilidad de usar Google Blockly, ya que el módulo de programación visual del robot había sido desarrollado con esta tecnología. La idea era que tras exportar los módulos de creación y simulación a navegadores (ambos módulos debidamente integrados), se llevara a cabo la comunicación entre los módulos de Unity y el de Blockly de forma más eficiente y en tiempo real por medio de algún tipo de intérprete.

Finalmente se descartó esta posibilidad, ya que existía una gran lista de problemas e inconvenientes a esto.

- La versión web requiere un complemento de terceros.
- No se admiten funciones de Unity como las corrutinas.
- La flexibilidad de diseño de interacción de la interfaz de usuario (UI) es baja.
- El coste de tiempo para realizar esta comunicación es bastante elevado. Además, no se puede garantizar un resultado óptimo, ya que la comunidad reporta unos grandes tiempos de espera entre el envío e interpretación de mensajes.

Por estos motivos decidí continuar investigando para finalmente encontrar UBlockly. Esta librería conseguiría ahorrar mucho trabajo y hacer que el proyecto fuera más sólido, ya que todo estaría desarrollado únicamente con Unity, y la comunicación entre el robot y los bloques sería interna, implicando que fuera instantánea.

UBlockly cuenta con tres módulos que se comunican entre sí. Estos serían *Code, Model* y *UI* (**Figura 8**).

- *Code:* Este módulo es el núcleo de funcionamiento de UBlockly. Tras la creación de nuestro script visual, permite **generar** el código C# fruto de la unión de los bloques del área de trabajo. Ya que C# es un lenguaje estático que no permite generar código de forma dinámica, requiere **interpretar** el código asociado al bloque como una implementación de C#. Esta interpretación será por medio del uso de *IEnumerator*, ya que consigue que el rendimiento

en ejecución no se vea limitado, además de que permite la reentrada al código, necesaria por la conectividad entre los bloques. Por último, tendríamos la parte de **ejecución,** que haría que el código empezara a funcionar. Hay algunos pequeños inconvenientes a raíz del uso de *IEnumerator*, entre ellos que no se puede pausar y reanudar la ejecución. Esta situación nosotros no nos supone ningún inconveniente, ya que solo vamos a necesitar parar la ejecución sin guardar el estado en el que se encontraba [24].

- *Model:* Constituido por el espacio de trabajo que contiene las variables y los bloques.
    - **Variables**: Son globales a todo el espacio de trabajo.
    - **Bloques**: Representan a un programa ejecutable, que pueden tener (o no) una salida, de forma análoga a una función en programación tradicional. Un bloque debe tener, al menos, un tipo de conexión para que se pueda conectar con otro bloque. Estas conexiones pueden ser de entrada, salida, anterior y posterior. Las conexiones entrada – salida permiten que un bloque reciba un parámetro a través de otro bloque, que, tras su ejecución, genera un valor de salida. Por otro lado, las conexiones anterior – posterior indican el orden de ejecución de estos. Los bloques son definidos en ficheros *.json* con su estructura y propiedades, pudiendo especificar su color, o si requiere que se introduzca un dato de entrada, como un número o texto [23].

- *UI:* Se ocupa de gestionar toda la parte visual con la que interactuará el usuario. Este módulo se encarga de crear los *Prefabs* de los bloques a partir de la estructura definida en los *.json*, o modificar el tamaño de estos en el área de trabajo en tiempo de ejecución si fuera necesario, entre otras funciones [25].

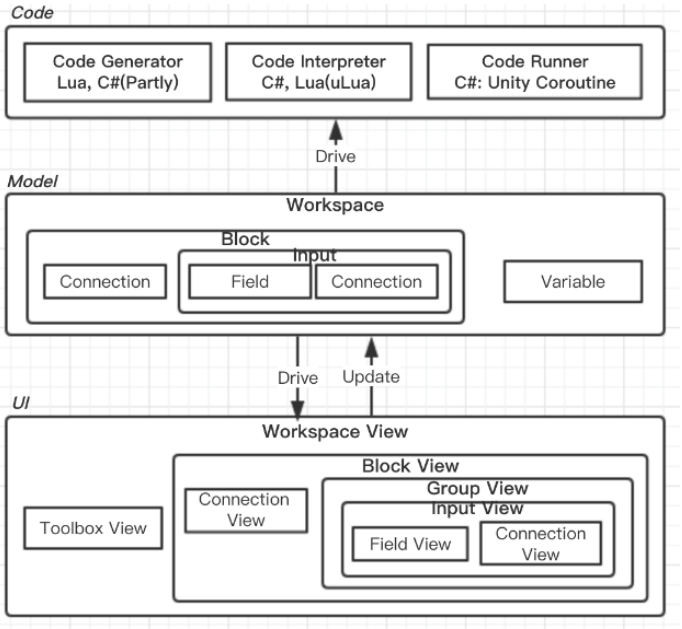


Figura 8: Esquema de funcionamiento de UBlockly

Desgraciadamente, más allá de la bibliografía ofrecida al final de este documento, no existe más información acerca de esta librería y la manera de integrarla en nuestros proyectos de Unity. Esta situación ha añadido algo de dificultad a todo este proceso. A pesar de ello, lo consideramos la alternativa más viable, ya que descubrir la forma de usarla fue simplemente cuestión de familiarizarse con la librería.

Para añadir UBlockly a nuestro proyecto, el primer paso será introducir el contenido de la librería dentro de la carpeta de nuestro proyecto. Concretamente la he ubicado en el directorio `Assets/Plugins`.

De serie UBlockly tiene varias categorías de bloques creadas, como los condicionales o los bucles. Para poder aprovechar las posibilidades de esta librería en nuestro proyecto, se crearon bloques específicos orientados a la ejecución de las acciones y recopilación de la información de los sensores del robot.

En las **Figuras 9 y 10** se muestra un pequeño ejemplo de cómo se crea un bloque en UBlockly:

```
{
    "type": "move_robot_forward",
    "message0:"%{BKY_MOVE_ROBOT}
                %{BKY_MOVE_ROBOT_FORWARD}
                con velocidad %1",
    "args0": [
        {
            "type": "field_number",
            "name": "DISTANCE",
            "value": 1,
            "min": 1,
            "max": 5,
            "int": true
        }
    ],
    "previousStatement": null,
    "nextStatement": null
},
```

Figura 9: Fichero de definición de bloques

```
"MOVE_ROBOT": "Mover robot",
"MOVE_ROBOT_FORWARD": "hacia delante",
```

Figura 10: Fichero de diccionario

- **type**: Apartado en el que nombraremos el nombre del bloque que vayamos a definir.
- **message0**: Constituye el mensaje que verá el usuario escrito dentro del bloque. Puede contener menciones a diccionarios, cuya sintaxis es un símbolo de porcentaje "%" junto con "BKY_ " y el nombre del elemento del diccionario. Todo esto debe ir contenido entre llaves "{}". También podemos poner argumentos con un "%" seguido de un número, que será una referencia al índice de este.
- **args**: Es la sección en la que definiremos los argumentos que tiene nuestro mensaje. Existen varios tipos de campos para los distintos tipos de dato, como pueden ser numérico, booleano, cadena de caracteres o listas con distintas opciones. A cada argumento se le asigna un nombre y posteriormente las distintas opciones específicas de cada tipo de campo.
- **previousStatement** y **nextStatement**: Con estas sentencias podemos especificar si el bloque tiene conexiones anterior o posterior.
- **output**: Indicamos que el bloque devuelve un dato, y el tipo de este dato.

Después de haber definido los bloques, debemos buscar en la barra de herramientas el botón "UBlockly". Tras clicarlo nos aparecerá la opción "*Build Block Prefabs*", que nos permitirá crear los *prefabs* de estos bloques (**Figura 11**).

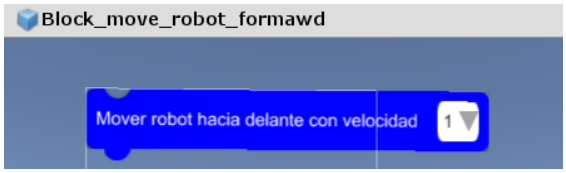


Figura 11: Ejemplo de bloque

3.2.3 GitHub

GitHub es una plataforma web que permite a los desarrolladores principalmente realizar control de versiones de sus proyectos y coordinar proyectos grupales de modo que se guarden e integren los cambios hechos por todos los colaboradores.

Además de usarse para realizar el control de versiones de este TFG, también se ha utilizado para poner a disposición de cualquiera el material desarrollado durante la realización de este trabajo, optando por subirlo a un repositorio público alojado en esta plataforma.

A continuación, podemos ver el enlace que nos llevará a este repositorio [29].

https://github.com/alu0100891843/UBlocklyRobot

Aprovechando una funcionalidad de GitHub llamada GitHub Pages he subido mi proyecto exportado a navegadores a un repositorio, en el que se podrá probar la aplicación [30].

https://alu0100891843.github.io/UBlocklyRobot_web/

3.2.4 Módulos de la aplicación

Se han aprovechado algunos de los recursos del proyecto desarrollado el anterior curso académico.

En cuanto al módulo de creación del robot, se ha podido utilizar prácticamente en su totalidad, ya que solo se eliminaron dos funcionalidades. En primer lugar, la exportación del fichero XML con la configuración del robot, debido a que no es necesaria. En segundo lugar, la personalización del tamaño del chasis y ruedas, ya que podría entorpecer el desempeño del autómata en la simulación.

Respecto al módulo de simulación hemos aprovechado bastantes partes. Aun así, se han tenido que descartar algunas funcionalidades, como la de lectura de los ficheros de instrucciones y de diseño del robot al no ser de utilidad. También debemos mencionar que no se ha incluido la librería *Jurassic* [28], ya que no necesitamos interpretar código JavaScript. Tampoco se ha aprovechado código relacionado con los hilos de los bloques, cuya función era conseguir un correcto ciclo de ejecución del código generado por la unión de los bloques, ya que el intérprete de UBlockly soluciona este problema.

Por último, respecto al módulo de programación visual, no se ha podido usar nada de material, debido a la decisión que se tomó de no continuar usando Google Blockly. A pesar de esto, se ha tenido en cuenta el diseño de las instrucciones de los bloques para este proyecto.

3.3 Desarrollo del proyecto

En este punto explicaremos cómo se ha llevado a cabo la fusión de todos los módulos para integrarlos en una única aplicación.

En primer lugar, se detalla la unión de los dos módulos desarrollados en Unity, consiguiendo que el robot pase de la escena de creación a la de simulación sin necesidad de cargar ficheros. En segundo lugar, la integración de UBlockly en el proyecto y la creación de los bloques necesarios para crear los scripts de definición del comportamiento. En tercer lugar, explicaremos las mejoras que se han añadido a la aplicación, además de los retos planteados para el entorno de simulación. Por último, se especifica el proceso seguido para hacer el proyecto compatible con los navegadores.

Antes de entrar en detalles técnicos, vamos a explicar cómo se relacionarán los módulos entre sí y qué funcionalidades tendrá la aplicación para tener una visión general de su funcionamiento.

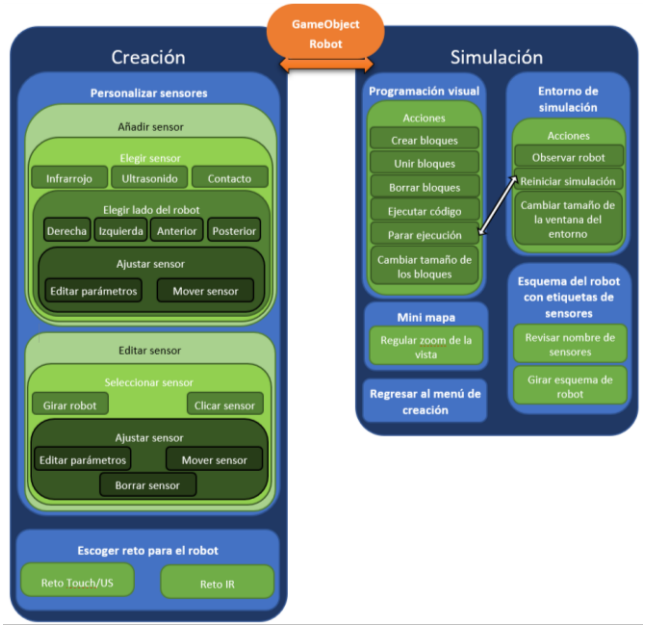


Figura 12: Esquema de funcionalidad y relación de los módulos actuales

Como vemos en la **Figura 12** tenemos dos partes bien diferenciadas. Estas serían la creación del robot y la simulación.

Respecto a la creación podemos editar nuestro robot personalizando sus sensores. Por una parte, se podrá añadir un nuevo sensor, donde tendremos que escoger cuál vamos a incorporar a nuestro diseño y elegir en qué lado del robot se ubicará. Cuando lo hayamos seleccionado podremos ajustarlo, tanto cambiando sus parámetros como localizándolo en una zona concreta del lado que hemos escogido para instanciar el sensor.

Por otra parte, podemos editar un sensor que ya haya sido creado buscándolo en el cuerpo del robot. Podemos girar el robot para encontrarlo y hacer clic sobre este. Cuando lo hagamos se podrá ajustar el sensor exactamente igual que al crearlo, además de poder borrarlo si nos hemos equivocado al añadirlo. No se ha incluido la funcionalidad de cambiar las dimensiones de la carrocería y las ruedas, ya que podía traer problemas a la hora de simular el robot.

En el momento en el que tengamos listo el robot debemos seleccionar uno de los dos retos disponibles. Al decidirnos por alguno de estos, se cargará la escena de simulación, en la que podremos ver varios elementos en pantalla.

Uno de los más destacables es el área de trabajo para la programación visual, en la que será posible definir el comportamiento del robot creando bloques, uniéndolos y ejecutando el código que desarrollemos con la ayuda de un esquema del robot que hemos diseñado. Este esquema sería una copia del diseño de nuestro autómata con las etiquetas de los sensores visibles. Junto con el área de trabajo estaría la ventana de simulación, en la que veremos al robot en el entorno cumpliendo las órdenes programadas con los bloques. Esta ventana cuenta con un mini mapa que nos da una perspectiva aérea del robot.

Existen algunas funcionalidades en este simulador que enriquecerán la experiencia de usuario, como por ejemplo la posibilidad de cambiar el tamaño de los bloques del área de trabajo, ya sea porque no podamos verlos con claridad o porque nuestro script tiene demasiados bloques y necesitamos hacerlos más pequeños para ganar espacio. También podemos alterar el tamaño de la ventana del entorno de simulación, de modo que podemos cambiar a pantalla completa siempre

que lo deseemos. Aunque una de las más interesantes podría ser la capacidad de reiniciar la simulación y detener la ejecución de los bloques, lo que da a los usuarios que han cometido algún fallo en el desarrollo del script la oportunidad de volver al inicio y corregirlo.

Cuando se finalice la simulación, podremos volver a la escena de creación para modificar nuestro robot y prepararlo para resolver el siguiente reto. Cabe destacar que el robot pasa automáticamente de una escena a otra sin ningún tipo de acción por parte del usuario.

3.3.1 Unión de los módulos de Creación y Simulación

El primer paso en el desarrollo de este TFG fue la unión de los módulos de creación y simulación. Al estar ambos desarrollados en Unity eran, por así decirlo "compatibles". El resultado final que se buscaba de dicha integración era que se pudiera crear el robot en la escena de creación y que este pasara directamente a la de simulación sin tener que exportar ni cargar ningún fichero.

Para comenzar, se unió el contenido de las carpetas de ambos módulos para contar con los *assets*, *prefabs*, escenas, scripts y demás recursos en un solo proyecto de Unity.

Después de hacer esto, hubo que corregir algunos problemas con incompatibilidades entre la versión de Unity de este TFG y la que había sido usada para el desarrollo de los proyectos que debía unir. Una de estas incompatibilidades era con *TextMeshPro*, que es un *asset* que permite una renderización en alta calidad del texto en las escenas. La incompatibilidad que surgió fue a causa de que para disponer este *asset*, los desarrolladores de los módulos tuvieron que incluirlo manualmente al proyecto descargándolo de la *Asset Store*, ya que estaban usando una versión anterior a la Unity 2018.2. A partir de esta versión viene integrado de serie, sin la necesidad de añadir ningún paquete adicional a nuestros proyectos.

Al descubrir esto, me dispuse a eliminar todo rastro del paquete *TextMeshPro* de las carpetas de *assets*. A continuación, hubo que actualizar todos los textos que estuvieran en las escenas, cambiándolos al *TextMeshPro* integrado. Además, se tuvieron que modificar las referencias que había en el código a estos textos, ya que se había cambiado la nomenclatura para nombrar a este tipo de elemento.

El siguiente paso fue conseguir que el GameObject del robot pasara de una escena a otra sin el sistema de carga de ficheros. Documentándome encontré un método conocido como "*DontDestroyOnLoad*", que permitía no destruir el GameObject y su contenido cuando se cambiara de escena. Para conseguir mi objetivo creé dos *prefabs*, uno del robot estándar, añadiéndole todos los scripts que necesita para ambas escenas y otro llamado "Permanente", que contiene un script que permite que no se borre al pasar de escena. Coloqué el GameObject "Permanente" en la escena de Creación con el GameObject del robot como hijo, y en la escena de Simulación coloqué "Permanente" sin ningún GameObject anidado (**Figuras 13** y **14**).

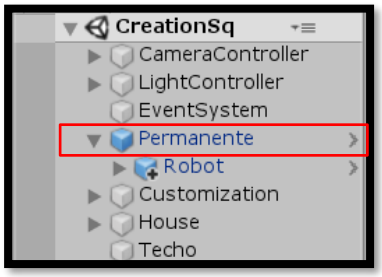


Figura 13: Permanente en la escena de creación

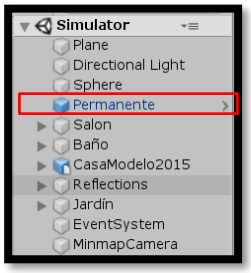


Figura 14: Permanente en la escena de simulación

También se habilitó un botón que permite el cambio de escena (**Figura 15**). Cuando conseguí que el paso del robot al entorno de simulación funcionara, tuve que llevar a cabo algunas pequeñas modificaciones a nivel de código. Añadí al robot un pequeño script **RobotScriptsController.cs,** el cual hace que algunos scripts del robot estén habilitados o deshabilitados dependiendo de la escena que esté activada. Los ficheros que se controlan son **Rotation.cs**, que permite al robot girar en la escena de creación usando las flechas y **RobotCollider.cs**, que cambia al robot de color en la escena de simulación cuando entra en contacto con una moneda situada en el entorno, además de activar o desactivar la cámara que usa el robot en la escena de simulación.

La activación y desactivación de estos scripts es crucial, ya que permite controlar las funciones que puede hacer el robot y alterar parámetros sus parámetros al cambiar de escena mediante la función "*Start*", que se ejecuta solo la primera vez que se habilita el script, y "*OnEnable"*, lanzada con cada activación del script. Algunos de los parámetros que se modifican son la posición, rotación y escala del robot. También se hacen ajustes relacionados con las físicas de las ruedas por medio de la propiedad *isKinematic* de los *Rigidbody* de esos objetos o el tamaño de la cámara.

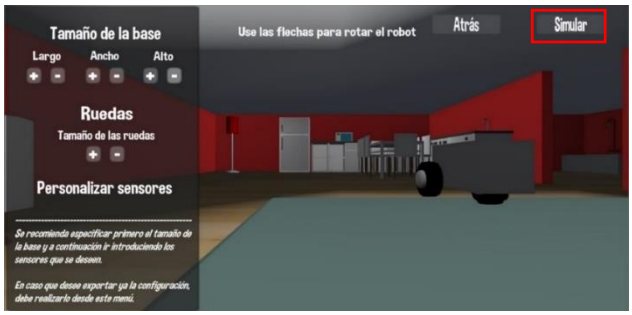


Figura 15 : Botón de comienzo de simulación

Por último, se tuvo que hacer algunas correcciones en las colisiones entre el robot y los sensores en la escena de simulación. El problema surgió por el hecho de que parte del sensor se encuentra dentro del modelo 3D del robot, generando una colisión continua entre ambos objetos. La solución consistió en deshabilitar la malla de colisión en las partes del sensor que colisionan con el robot, lo que no afecta al funcionamiento del sensor (**Figura 16**).

Figura 16: Sensor láser colisionando por la parte trasera con el modelo 3D del robot

El resultado final de la estructura de carpetas tras unir los módulos es el siguiente (**Figura 17**):

- **Carpetas con recursos de la *Asset Store* heredados del anterior proyecto**:
  - ***AxeyWorks*:** Elementos decorativos de exterior**.**
  - ***BigFurniturePack*:** Elementos decorativos de interior, como es el mobiliario.
  - ***Casa*:** Modelo de la casa del entorno de simulación.
  - ***ConeCollider*:** Modelo de colisión de forma cónica usado para el sensor de contacto.

- ***Ground textures pack y QS*:** Contienen texturas para el suelo, entre ellas una con efecto de césped que es la que se ha aprovechado.
- ***Wispy Sky*:** *Asset* que permite generar un cielo más realista en la escena de pruebas del robot.

- **Scenes**: Contiene las distintas escenas con las que cuenta el proyecto.
- **Scripts**: Almacena los scripts necesarios para la creación y simulación del robot. Hay scripts de propósitos varios, aunque buena parte de estos está orientada al control de los sensores. Por este motivo, se decidió crear un subdirectorio que los separe del resto.
- **Prefabs**: Almacena los *prefabs* de todos los elementos que componen al robot (chasis y ruedas), además de los distintos sensores que podemos acoplar a este.
- **Images**: Contiene las imágenes que se usan en el proyecto.
- **Resources:** En su interior hay una colección de los recursos, tanto texturas como *prefabs* más usados provenientes de la *Asset Store*.

Figura 17: Estructura de directorios conjunta de los módulos de simulación y creación

3.3.2 Unión de los módulos de Unity con el módulo de programación visual

El siguiente paso tras haber logrado unir con éxito los módulos de Unity era integrar de algún modo el módulo de programación visual. Como se explicó en el punto 3.2.2 de esta memoria, en un principio se pretendía comunicar el módulo de programación visual desarrollado en el anterior curso académico con los módulos de Unity en el navegador. Esta opción contaba con bastantes inconvenientes, por lo que se decidió descartarla. Tras realizar una tarea de investigación, encontré un proyecto conocido como UBlockly, que permitía integrar Blockly directamente en Unity.

Lo primero que hice fue probar unas demos que el desarrollador de esta librería había dejado para ver su funcionamiento. Afortunadamente era justo lo que necesitaba, así que me decidí a integrar UBlockly dentro de mi proyecto. Este proceso es bastante sencillo, aunque tuve algunas dificultades para saber cómo llevarlo a cabo, ya que la documentación de esta herramienta no es tan detallada como debería. Con ayuda de las demos que había descargado y examinando la estructura de carpetas y ficheros logré descubrir cómo hacerlo.

Para comenzar con el proceso de integración, descargué la librería de UBlockly desde GitHub, para añadir los ficheros a nuestro proyecto, dando como resultado esta estructura (**Figura 18**):

- **AllRes/Robot**: Contiene tres subdirectorios que almacenan los ficheros .json con los que se definirán los bloques que se usarán. Estos serían el fichero de diccionario y el que define la estructura que va a tener la caja de herramientas (*toolbox*) del área de trabajo de Blockly.
- **Plugins:** Almacena parte del contenido del repositorio de GitHub de UBlockly, concretamente fotografías de la documentación y una estructura simple para una *toolbox* que únicamente contiene las categorías básicas predefinidas de la librería junto con la definición de los bloques de cada categoría (una categoría es un conjunto de bloques que comparten una temática en su

funcionalidad). También almacena los dos tipos de diseño de *toolbox* disponibles, que son *ClassicToolbox* (la elegida para este proyecto) y *ScratchToolbox*.

- **ProjectData**: Al ejecutar la orden de creación de los *prefabs* de los bloques (después de haberlos definido en los ficheros .json), se almacenan en esta carpeta junto con el resto de *prefabs* del proyecto. No debemos olvidarnos de mencionar que también contiene dos ficheros muy importantes para la personalización de los bloques que utilizaremos en la *toolbox* del entorno de simulación. Estos dos ficheros son **BlockResSettings** y **BlockViewSettings**.
- **Scripts/UToolBox:** Los scripts de esta carpeta contienen las instrucciones necesarias para almacenar y cargar scripts de bloques hechos por el usuario.
- **Scripts/Common:** Estos scripts permiten la ejecución ordenada y correcta de todas y cada una de las órdenes de los bloques con mecanismos de finalización de *frame* o espera de condiciones.
- **Scripts/Robot:** En esta carpeta están los ficheros que permiten asignar un fragmento de código a los bloques que hemos definido.
- **Source:** Contiene los intérpretes, generadores y lanzadores de código. También cuenta con las definiciones de los distintos cuadros de diálogo, gestores de las conexiones entre bloques y otros ficheros encargados del funcionamiento, gestión y correcta visualización del módulo de programación por bloques.

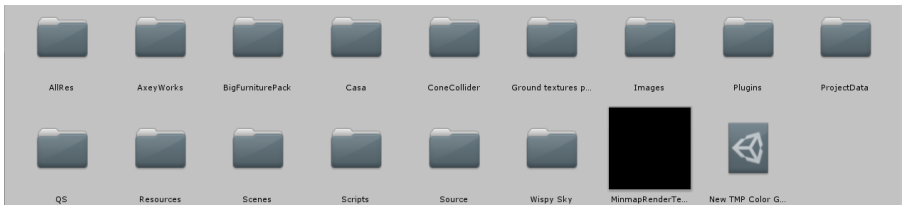


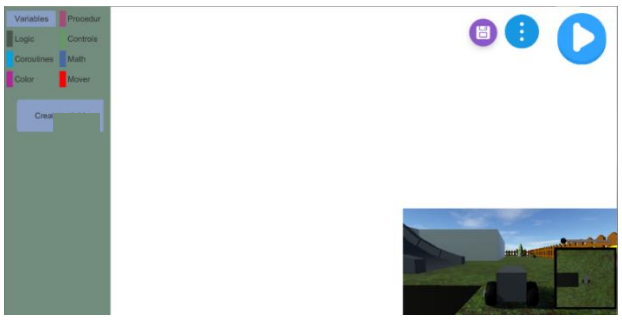
Figura 18: Estructura de directorios con UBlockly integrado

Una vez hecho esto, creé el área de trabajo dentro de la escena de simulación a partir de un *canvas* (es un GameObject que permite renderizar elementos destinados a la creación de la interfaz de usuario) que contiene un *prefab* de la librería llamado "Workspace" (**Figura 21**). Podemos ver que existe una zona en la que se pueden ver las listas de colecciones de bloques, el área de codificación y algunos botones que nos permiten ejecutar, guardar y cargar nuestros scripts (**Figura 20**).

Para poder visualizar tanto el área de trabajo como el entorno de simulación se llevaron a cabo algunos ajustes en la cámara. En concreto, en el GameObject "Robot Camera", que es la cámara que apunta al robot, modificando sus valores en el parámetro "Rect" al cambiar a la escena de simulación. De este modo, se visualizará el robot en una esquina para poder trabajar cómodamente en el espacio de trabajo de UBlockly. En la **Figura 19** está el fragmento de código del fichero **RobotCollider.cs** que se activa al cambiar a la escena de simulación.

```
    void Start(){
        main = GameObject.Find("Robot Camera").GetComponent<Camera>();
        main.rect = new Rect(0.65f,-0.45f,1,0.8f);
    }
```

Figura 19: Cambio de tamaño de la ventana del entorno de simulación



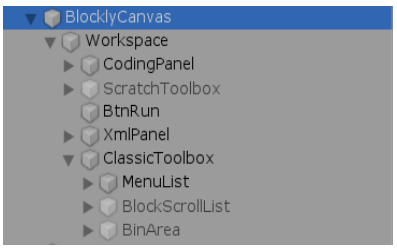


Figura 20: Escena de simulación con UBlockly integrado

Figura 21: Estructura del Workspace

Como podemos apreciar en la **Figura 20**, la zona de *toolbox* contiene varias categorías, que son las que vienen por defecto si no hemos editado nada de **BlockResSettings**. Estas categorías cuentan con muchos bloques útiles, con los que podremos crear bucles, condicionales, operadores de comparación, operadores matemáticos, etc.

El siguiente objetivo fue crear nuestras propias categorías. La primera que se añadió se denominó como "MOVE". Esta debía contar con las órdenes que el robot iba a desempeñar, que serían desplazarse hacia delante o hacia atrás y girar.

Para definir estas acciones, en primer lugar, se tuvieron que crear tres nuevos ficheros en la carpeta **AllRes/Robot**. El primero que creamos fue **moveBlocks,** en el que se definirán los bloques de la categoría "MOVE". El segundo es **move_en**, que contiene el diccionario con las palabras que usarían los mensajes de los bloques. El último sería **toolbox_robot**, donde se definen las categorías y los colores que tendrán los bloques de estas (**Figuras 22**, **23** y **24**).

Las categorías que contendrá por el momento este *toolbox* son las predefinidas más la categoría que vamos a añadir.

```
{
    "Style": "classic",
    "BlockCategoryList": [
        {
            "CategoryName": "VARIABLE",
            "ColorHex": "#8a9fc6",
            "BlockTypePrefix": "variables"
        },
        //Resto de categorías predeterminadas
        {
            "CategoryName": "MOVE",
            "ColorHex": "#ff0000",
            "BlockTypePrefix": "move"
        },
    ]
}
```

Figura 22: Fichero definición de toolbox (toolbox_robot)

```
{
    "type": "move_robot",
    "message0": "%{BKY_MOVE_ROBOT}
                %{BKY_MOVE_DISTANCE}
                %1",
    "args0": [
        {
            "type": "field_number",
            "name": "DISTANCE",
            "value": 0,
            "min": 0,
            "max": 10,
            "int": true
        }
    ],
    "previousStatement": null,
    "nextStatement": null
}
```

Figura 23: Fichero definición de bloque (moveBlocks)

```
{
    "MOVE": "Mover",
    "MOVE_ROBOT": "Mover robot",
    "MOVE_DISTANCE": "a velocidad"
}
```

Figura 24: Fichero diccionario (move_en)

En segundo lugar, se editó el fichero **BlockResSettings**, situado en la carpeta `ProjectData`. Para ello debemos buscarlo en el explorador de Unity y hacer doble clic. Con esta acción se mostrará

una ventana en el inspector de Unity con muchos parámetros que podemos modificar. Ahora debemos arrastrar los tres ficheros que hemos creado a la zona que corresponda. En "I18n Files" añadimos el fichero de diccionario, en "Block Json Files" el fichero de definición de bloques y en "ToolBox Files" el de definición de *toolbox*. Además, debemos añadir la ruta en la que se guardarán los *prefabs* de los bloques. En nuestro caso el directorio sería `ProjectData/BlockGenPrefabs`.

Las **figuras 25**, **26** y **27** tendrían la configuración resultante de las modificaciones que hemos llevado a cabo en el fichero **BlockResSettings**:

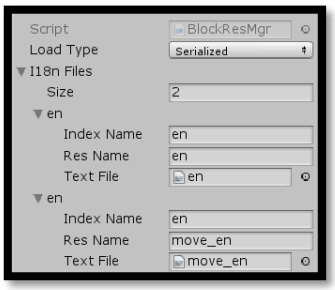


Figura 25: BlockResSettings diccionario

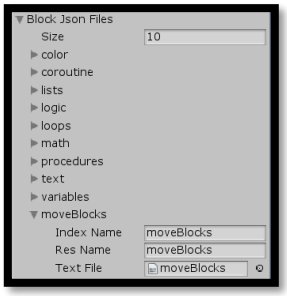


Figura 26: BlockResSettings bloques

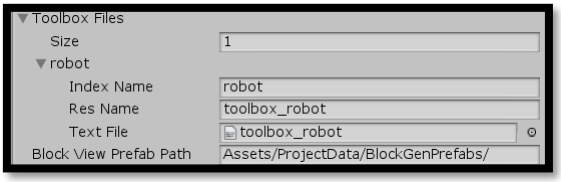


Figura 27: BlockResSettings toolbox

La configuración que hemos establecido dará como resultado al ejecutar la acción de construir los bloques el siguiente *prefab* de bloque de la categoría "MOVE" (**Figuras 28** y **29**):

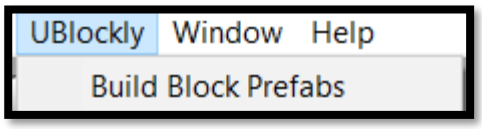


Figura 28: Construir bloques

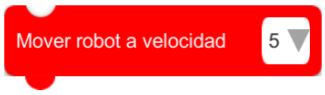


Figura 29: Bloque de movimiento

Hasta el momento había conseguido crear el *prefab* de un bloque, pero este al ser ejecutado no hace nada, ya que no tiene ningún fragmento de código asociado, por lo que debíamos asignarles scripts a estos bloques.

Para ello me dirigí a la carpeta `Scripts/Robot` y creé dos ficheros con extensión C#, **RobotCmds** y **RobotController**.

En el archivo **RobotCmds** será donde asignaremos la función que hará un determinado bloque. Si hubiese que definir algún parámetro en este, se recogen los valores de esos campos para a continuación, pasarlos a otra función que estará en el fichero **RobotController**, donde se definirá el comportamiento del bloque.

Las **figuras 30** y **31** nos muestran el código de una primera versión de los ficheros de asignación y definición de comportamiento del bloque de movimiento hacia delante.

```
public class Move_Robot_Cmdtor : EnumeratorCmdtor
    {
        protected override IEnumerator Execute(Block block)
        {   string distanceStr = block.GetFieldValue("DISTANCE");
            int distance = int.Parse(distanceStr);

            yield return RobotController.Instance.DoMove(distance);
        }
    }
```

Figura 30: Fichero de asignación de función (RobotCmds)

```
public IEnumerator DoMove(int distance){
    robot.GetComponent<Rigidbody>().AddForce(Vector3.forward * distance * 100);
    yield return null;
}
```

Figura 31: Fichero de definición de comportamiento (RobotController)

Esta primera versión no es la mejor para ejecutar un movimiento controlado, ya que si observamos detenidamente el código de **RobotController**, estamos aplicando una fuerza de empuje que aunque tenga una intensidad que podemos ajustar, puede llevar a un movimiento impreciso del autómata. A pesar de todo, este paso supuso un antes y un después en el proyecto, ya que poder ejecutar el bloque correctamente nos indicó que todo estaba funcionando según lo esperado. Además, el hecho de haber descubierto como crear este bloque permitiría crear otros fácilmente siguiendo la misma metodología.

Después de cumplir este objetivo, debíamos crear el resto de bloques que fueran necesarios para la categoría de movimientos del robot y mejorar el sistema de avance de este. Las acciones que se habían planeado para los movimientos del robot son las que enumeramos en la siguiente lista:

- Avance y retroceso con velocidad variable durante un periodo de tiempo.
- Avance y retroceso con velocidad variable con tiempo indefinido.
- Giro hacia la derecha o izquierda con grados ajustables.
- Giro hacia la derecha o izquierda indefinido.
- Detenención o parada.

Durante la investigación acerca de cómo mejorar el sistema de avance, descubrí que las ruedas del robot tienen un componente llamado *HingeJoint*, que sirve para crear una unión entre las ruedas y la carrocería del robot, permitiendo que puedan girar y hacer que el autómata se desplace. Dicho componente cuenta con un parámetro denominado “motor”, que permite hacer que la rueda gire en ambos sentidos y a distintas velocidades. Gracias a este descubrimiento, rediseñé la función de avance del robot, sustituyendo la fuerza de empuje por un accionamiento de los motores. También añadí unas restricciones en la rotación que pueden tener las ruedas durante este avance, lo que favorece que sea lo más recto posible. Para concluir, se hicieron unos ajustes en el rozamiento de estas para que no hagan efecto de derrape al desarrollar el movimiento (**Figura 32**).

```
public IEnumerator DoMoveForward(int distance,int time){

            robot_rb.constraints = RigidbodyConstraints.FreezeRotation;
            wheel_r_rb.drag = 5; wheel_l_rb.drag = 5; wheel_c_rb.drag = 5;
             //Asignación de rozamiento y restringiendo rotación innecesaria de las ruedas
            JointMotor motor_r,motor_l,motor_c;
            motor_r = wheel_r.motor; motor_l = wheel_l.motor; motor_c = wheel_c.motor;
```

```
            motor_r.targetVelocity = 300*distance;
            motor_l.targetVelocity = -300*distance;
            motor_c.targetVelocity = -300*distance/3;

            motor_r.force = 200*distance;
            motor_l.force = 200*distance;
            motor_c.force = 200*distance/3;

            wheel_r.motor = motor_r;
            wheel_l.motor = motor_l;
            wheel_c.motor = motor_c;
            //Asignación de potencia y velocidad a las ruedas
            wheel_r.useMotor = true; wheel_l.useMotor = true; wheel_c.useMotor = true;
             //Accionando los motores
        if (time != 0) {//Condición para el bloque que tenga tiempo definido
                yield return new WaitForSeconds(time);//Para los motores al acabar el tiempo
                wheel_r.useMotor = false; wheel_l.useMotor = false; wheel_c.useMotor = false;
            }
            yield return null;
        }
}
```

Figura 32: Redefinición MoveForward (RobotController)

Este script sirve tanto para movimiento de avance indefinido como avance por un periodo de tiempo, ya que si es indefinido, se iguala el parámetro "*time*" de esta función a 0. El método de retroceso del robot es exactamente igual que este, pero cambiando los signos de las velocidades de las ruedas.

En cuanto al desarrollo de las funciones para el giro de las ruedas, se aprovechó que cada una tenía un motor independiente. Por ejemplo, si hacemos que la rueda derecha gire hacia delante y la izquierda hacia atrás, conseguimos un giro del robot hacia la izquierda. Este script es bastante interesante, sobretodo las condiciones de parada del giro, que es donde centraremos la explicación.

```
...//Accionamiento de las ruedas para giro a la izquierda
      while (girando & angle != 12345){
        if (final_angle == robot.GetComponent<Transform>().eulerAngles.y) {
            girando = false;
        }
        if ((girando) & (final_angle <= epsilon || final_angle >= 360-epsilon){
            if (initial_angle > final_angle){ //Ángulo final cercano a 0 por la derecha
                if (robot.GetComponent<Transform>().eulerAngles.y - epsilon <= final_angle){
                    girando = false;
            }}
            else{ //Ángulo final cercano a 0 por la izquierda
                if (robot.GetComponent<Transform>().eulerAngles.y - epsilon >= final_angle){
                    girando = false;
        }}}
        if ((girando) & (robot.GetComponent<Transform>().eulerAngles.y < final_angle) &
           (robot.GetComponent<Transform>().eulerAngles.y <= initial_angle) & (!inverted))
        {}
        else{
            if ((girando) & (robot.GetComponent<Transform>().eulerAngles.y > final_angle)){
                inverted = true;
            }
            else{
                girando = false;
        }}
        yield return new WaitForEndOfFrame();
    }
    if (angle != 12345){ //Corrección del ángulo final
        Transform fix_turn = robot.GetComponent<Transform>();
        fix_turn.eulerAngles = new Vector3(fix_turn.eulerAngles.x,final_angle,fix_turn.eulerAngles.z);}
    }
...//Detención de los motores de las ruedas
```

Figura 33: Condiciones de parada de giro a la izquierda (RobotController)

En concreto, el fragmento de código de la **Figura 33** es para la condición de parada de giro a la izquierda, muy similar a la de giro a la derecha, en la que cambian algunos símbolos en los condicionales.

Parte de este código está inspirado en un script que ya había sido desarrollado, aunque este no era del todo correcto, ya que, bajo algunas circunstancias, provocaba fallos con ángulos finales cercanos a 0º y no detenía el giro del robot, por lo que hubo que hacerle algunas modificaciones.

En este script hacemos varias comprobaciones para determinar si debemos detener el giro. La primera y la condición más sencilla, si el ángulo actual es el ángulo objetivo, se paran las ruedas. Si no se cumple, comprueba si el ángulo final es cercano a 0 tanto por la izquierda como por la derecha. De ser así, se detendría el giro. En el caso de que tampoco se cumplan estos condicionales, se verifica si el ángulo de giro del robot es menor que los ángulos final e inicial y no está activado un booleano *inverted*. Si no satisface la condición, comprueba que el ángulo del robot es mayor que el final. En el caso de que no se cumpla, se detiene el robot. Después de la ejecución del bucle, se realiza una corrección del ángulo de giro por si al detener las ruedas se llegaran a mover por la inercia del movimiento.

Estas condiciones de parada únicamente se dan cuando se marca la cantidad de grados que se debe girar. En el caso de que no se haga, tan solo se activará el giro sin frenar las ruedas.

Dicho esto, con este código conseguí crear los giros a ambos lados, tanto con grados ajustables como indefinidos, por lo que ahora quedaría crear un bloque para detener el movimiento del robot. Este bloque tiene una implementación bastante simple, ya que lo que se hace es desactivar el motor y aumentar el rozamiento de las ruedas hasta que no puedan seguir rotando (**Figura 34**).

```
public IEnumerator DoStop(){
    robot_rb.constraints = RigidbodyConstraints.FreezeRotation;

    wheel_r.useMotor = false; wheel_r_rb.drag = 9999999;
    wheel_l.useMotor = false; wheel_l_rb.drag = 9999999;
    wheel_c.useMotor = false; wheel_c_rb.drag = 9999999;
    yield return new WaitForSeconds(1);
}
```

Figura 34: Código de detención de movimiento

Al haber concluido la definición del comportamiento de los bloques, y por supuesto, haber creado la estructura de estos debidamente para crear los *prefabs* de estos, tal y como se hizo con el primer bloque de movimiento, obtendríamos los bloques que podemos ver en la **Figura 35**.

Figura 35: Repertorio de bloques de la categoría MOVE

Después de hacer varias pruebas para corroborar que funcionaban adecuadamente, llegó el turno de crear los bloques de los sensores, que nos permitirán condicionar las acciones del robot en función de la información que estos recojan del escenario. Para este TFG se han implementado

bloques de los sensores de ultrasonido, contacto e infrarrojo, descartando el uso del sensor y telémetro láser, ya que el sensor ultrasonido puede sustituir a ambos.

Con el fin de seguir una organización en la estructura de la caja de herramientas, se decidió crear otra categoría para almacenar estos nuevos bloques. El método para crear esta categoría es idéntico al que usamos para instanciar la categoría "MOVE". Creamos dos ficheros en **AllRes/Robot**, **sensorBlocks** y **sensor_en**, que almacenarían la definición de los bloques y el diccionario respectivamente. También hubo que editar el fichero **robot_toolbox** para añadir la categoría "SENSOR", además de añadir a **BlockResSettings** los dos nuevos ficheros que hemos creado.

La creación de estos bloques tuvo algunas dificultades frente a los de movimiento. Por una parte, los sensores debían estar etiquetados de alguna forma para hacer referencia a un sensor específico en el bloque. Por otra parte, los bloques debían retornan los datos de los sensores.

Para solucionar el primer problema, decidí etiquetarlos cambiando su nombre al instanciarlos, de modo que se tuvieron que hacer unas modificaciones en las funciones "*Start*" de los ficheros **IRSensorScript**, **USSensorScript** y **TouchSensorScript**. En estos se añadirían un fragmento de código donde se buscan cuántos sensores hay del tipo que se pretende instanciar y se renombra el sensor con el tipo de este, seguido del número del sensor que corresponde a la búsqueda realizada (**Figura 36**). He de destacar que el funcionamiento de la búsqueda es el mismo para todos los sensores.

```
    void Start () {…//Código de definición del sensor
    //Búsqueda de sensores en el robot
       int count = 0;
       foreach (Transform side in go.transform) {
           foreach (Transform tipoSensor in side){
               if(tipoSensor.tag == "tipo"){
                   count++;
                   Debug.Log(count);
                   if (GameObject.Find("SensorTipo"+count.ToString())==null){
                   //Si no hay un sensor con este nombre
                   //Se para la ejecución y se asigna dicho nombre
                      break;
       }}}}
       gameObject.name = "SensorTipo"+count.ToString();
}}
```

Figura 36: Renombrar sensores instanciados para su etiquetado

Para dar solución al segundo problema de retorno de valores, decidí revisar la definición de algunos bloques predefinidos que eran parte de la librería. Finalmente conseguí encontrar varios ejemplos que me sirvieron para comprender cómo devolver datos con un bloque.

```
    [CodeInterpreter(BlockType = "sensor_us")]
    public class Detection_Sensor_US_Cmdtor : ValueCmdtor{
        protected override DataStruct Execute(Block block){
            int index = int.Parse(block.GetFieldValue("NUMBER"));
            double data = RobotController.Instance.INFOSensorUS(index);
            Debug.Log("Devuelve "+data.ToString());
            return new DataStruct(data); //Retorna el valor de la función
    }}
```

Figura 37: Función de asignación sensor ultrasonido (RobotCmds)

```
public double INFOSensorUS(int index){
    double distance;
    USSensorDistance us =
                        (USSensorDistance)GameObject.Find("SensorUS"+index).transform.
                         GetChild(16).GetComponent(typeof(USSensorDistance));
    if (us.getDetection ())  distance = (double)us.getDistanceHit ();
    else distance = double.MaxValue;
    return distance; }
```

Figura 38: Función de definición sensor ultrasonido (RobotController)

A diferencia de las funciones de movimiento, la función "*Detection_Sensor_US_Cmdtor*" del fichero **RobotCmds** (**Figura 37**) recoge el valor que retorna la función "*INFOSensorUS*" de **RobotController** (**Figura 38**) de tipo *double***,** que toma la información del sensor. Por último, "*Detection_Sensor_US_Cmdtor*" devolverá una estructura de datos que contiene el valor que nos ofrece el sensor para que el bloque ofrezca los datos recopilados por el sensor. El resto de las funciones de sensores son similares a las que acabamos de explicar.

Después de seguir los pasos correspondientes que detallamos anteriormente en este documento, por medio de los que podremos definir y crear el comportamiento de los bloques, obtendremos el siguiente conjunto de bloques para la categoría de los sensores. Esto dará como resultado la *toolbox* completa con todos los bloques necesarios para el control del robot (**Figura 39 y 40**).

Figura 39: Repertorio de bloques de la categoría

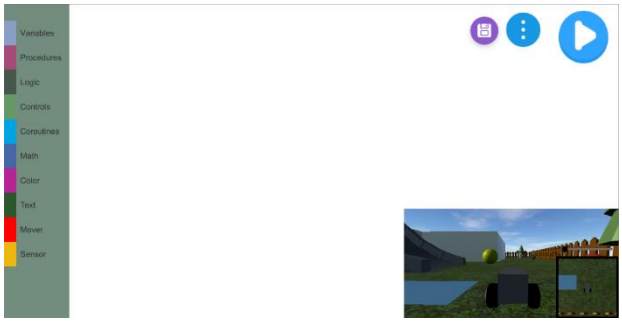


Figura 40: Escena de simulación con las categorías MOVE y SENSOR

Haber conseguido finalizar con éxito la creación de esta última categoría daría por concluida la parte de integración de los tres proyectos.

3.3.3 Mejoras de funcionalidades e interfaz

En este apartado se detallan todas las mejoras que se han hecho tanto a aspectos visuales como a nivel de funcionalidad para mejorar la experiencia de usuario.

Vista de las etiquetas de sensores

Una de estas mejoras fue mostrar las etiquetas de los sensores al usuario, para que este pudiera distinguir los sensores entre sí. Hubo que hacer modificaciones en los *prefabs* de los sensores añadiendo un texto que sería el que mostraría la etiqueta y escribir una línea en los ficheros de creación de sensores (**Figura 36**), en la que se iguala el texto de la etiqueta al nombre del sensor (**Figura 41** y **42**).

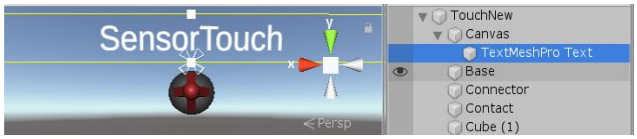


Figura 41: Añadiendo etiqueta de sensor

```
public TextMeshProUGUI sensorname;
...
gameObject.name = "SensorTipo"+count.ToString();
sensorname.text = "SensorTipo"+count.ToString();
```

Figura 42: Modificando texto de la etiqueta

Además, también se llevó a cabo una modificación que cambia de color esta etiqueta para indicarnos qué sensor estamos creando o editando. Para lograrlo, se tuvo que añadir algunas líneas que permiten alterar el color del nombre del sensor en el script de creación **CrearSensor** y en el que permite editarlos, llamado **OptionsSensorFront** (**Figura 43**).

```
//CrearSensor
    public void crearTouchBack() {
        GameObject touchGO = (GameObject)Instantiate (touchPrefab,placeBack.position,placeBack.rotation,parentBack);
        options.setGo (touchGO);
        touch = (TouchSensorScript) touchGO.GetComponent (typeof(TouchSensorScript));
        options.setUSScript (us);
        //Añadir las siguientes dos líneas a las funciones de creación de sensores Touch, IR y US
        text = GameObject.Find(touch.name + "/Canvas/TextMeshPro Text").GetComponent<TextMeshProUGUI>();
        text.color = new Color(255,255,0,255);
        ...
    }
//OptionsSensorFront
    void Update () {
      ...
      if (go.tag == "US" || go.tag == "Touch" || go.tag == "Laser" || go.tag == "IR" || go.tag == "Lidar"){
        text = GameObject.Find(go.name + "/Canvas/TextMeshPro Text").GetComponent<TextMeshProUGUI>();
        text.color = new Color(255,255,0,255);
     }}
    public void setSlidersLaserToGameObject(){
        ...
        if (go.tag == "US" || go.tag == "Touch" || go.tag == "IR"){
            text = GameObject.Find(go.name + "/Canvas/TextMeshPro Text").GetComponent<TextMeshProUGUI>();
            text.color = new Color(255,255,255,255);
        }
        go = null;
    }
```

Figura 43: Cambio de color de las etiquetas de los sensores

La primera función, localizada en el fichero **CrearSensor** serviría para cambiar el color al crear el sensor. En este caso, sería para el de contacto trasero, por lo que habría que añadir esta modificación para las demás funciones de creación. En la función "*Update*" del archivo **OptionsSensorFront** cambiamos el color a las etiquetas de los sensores que estamos tratando de modificar. En la tercera y última función, restablecemos el color de la etiqueta al original.

Por último, para que las etiquetas solo sean visibles en el menú de creación, esta es desactivada al pasar a la escena de simulación. Esto es posible gracias a los scripts **Rotation.cs** y **RobotCollider.cs**, ya que como se explicó en el punto 3.3.1, la activación y desactivación de estos permite cambiar parámetros y aspectos del robot dependiendo de la escena en la que nos encontremos. La activación de las etiquetas debería ir en la función "*OnEnable*" de **Rotation**, al ser el script que está habilitado en la escena de creación. El método que nos permite ocultar los nombres de los sensores debe estar ubicado en la misma función, pero en el fichero **RobotCollider,** que es habilitado en la simulación (**Figuras 44** y **45**)

```
GameObject sensor_id;
for (int i=2; i<7; i++){
   for (int j=2; j< gameObject.transform.GetChild(i).childCount; j++){
      sensor_id = gameObject.transform.GetChild(i).
                  GetChild(j).GetChild(0).gameObject;
      sensor_id.SetActive(false);
      //Para habilitar cambiar SetActive a true
}}
```

Figura 44: Desactivación de etiquetas (RobotCollider)

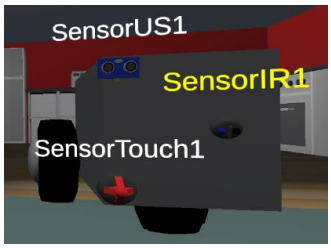


Figura 45: Etiquetas de sensores en el robot

Creación de esquema del robot en entorno de simulación

Con el fin de hacer más sencilla la tarea de recordar cómo era el diseño de nuestro robot, se pensó que sería una buena idea agregar una copia de este en la escena de simulación, de modo que sirviera de guía para consultar las etiquetas de los distintos sensores con los que cuenta nuestra creación. Al cambiar a la escena de simulación, se crea una copia del robot a la que se le hacen algunas modificaciones. Concretamente, tiene los ajustes del robot de la escena de creación, por lo

que puede girar si se pulsan las teclas de las flechas en el teclado, además de poder ver las etiquetas de los sensores. Esto se ha conseguido habilitando el script **Rotation** (**Figura 47**) y desactivando **RobotCollider**, además de deshabilitar **RobotScriptController** (**Figura 46**) para que no se hagan comprobaciones de en qué escena está este GameObject y altere los scripts que este puede usar.

Por último, en la función "*Start*" de **Rotation,** se ajusta la posición y rotación del robot, aparte de eliminar cualquier restricción que pueda tener el robot para rotar.

Con estas modificaciones se consiguió crear la copia del robot en la escena de simulación (**Figura 48**).

```
if (SceneManager.GetActiveScene().name == "Simulator"){
     if (GameObject.Find("Robot_Esquema") == null){
        var temp = Instantiate(gameObject);
        temp.name  = "Robot_Esquema";
        rscriptCTLRsq = temp.GetComponent<RobotScriptsController>();
        rotationCSsq = temp.GetComponent<Rotation>();
        rcolliderCSsq = temp.GetComponent<RobotCollider>();
        rscriptCTLRsq.enabled = false;
        rotationCSsq.enabled = true;
        rcolliderCSsq.enabled = false;
}}
```

Figura 46: Creación y configuración de scripts de Robot_Esquema (RobotScriptController)

```
    void Start(){
        if (gameObject.name == "Robot_Esquema"){
            robot_tr = gameObject.GetComponent<Transform>();
            robot_tr.localScale = robot_tr.localScale * 6;
            robot_tr.localPosition = new Vector3(10005.5f,10001,10054);
            robot_tr.localEulerAngles = new Vector3(0,180,0);
            robot_cp = gameObject.GetComponent<Rigidbody>();
            robot_cp.constraints = RigidbodyConstraints.None;
    }}
```

Figura 47: Cambio de posición, rotación y restricciones de Robot_Esquema (Rotation)

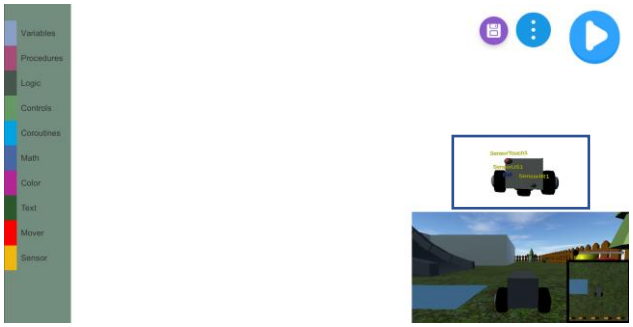


Figura 48: Esquema del robot en la escena de simulación

Por último, deberíamos mencionar que se modificó la fuente de las etiquetas de los sensores para que fuera más legible, optando por amarillo como color normal, y verde como color de sensor seleccionado en la escena de creación, además de acentuar los bordes de las letras.

Cambio de tamaño de la ventana del escenario de simulación

Pensamos que algunos usuarios querrían ver el entorno de simulación en pantalla completa para poder apreciar mejor el escenario y ver el recorrido del robot con más claridad. Por este motivo, se añadió un botón que permitiera alterar el tamaño de la ventana en tiempo de ejecución (**Figura 50**).

Hubo un pequeño problema para poder llevar esto a cabo, y es que al colocar el botón en la escena de simulación y querer buscar la cámara del robot no la encontraba, ya que este objeto era hijo de "DontDestroyOnLoad" al ser un objeto que no se destruye con el paso entre escenas . Por el contrario, el botón de cambio de tamaño pertenecía a la jerarquía de "Simulador". El hecho de pertenecer a tipos de escenas distintos impedía hacer la búsqueda. Por este motivo se añadió este botón al objeto "Permanente", de modo que así no ocasionaría problemas al buscar la cámara.

```
public void MinOrMax(){
        if (cam_.rect == new Rect(0.65f,-0.45f,1,0.8f)){
            //Se hace grande
            cam_.rect = new Rect(0,0,1,1);
        }
        else{
            //Se hace pequeño
            cam_.rect = new Rect(0.65f,-0.45f,1,0.8f);
}}
```

Figura 49: Cambio de tamaño de la ventana del entorno de simulación

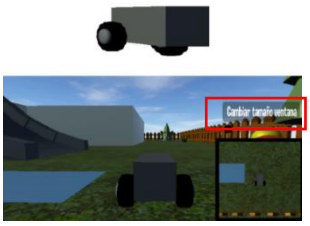


Figura 50: Botón de cambio de tamaño de ventana

La pulsación del botón de cambio de tamaño llama a la función "*MinOrMax*" del fichero **Maximize_On_Click** (**Figura 49**). Lo que hace este método es alternar el parámetro "Rect" de la cámara "Robot Camera", que tomará un valor u otro dependiendo del valor actual de este.

Reiniciar escena

Esta nueva funcionalidad dará la oportunidad de reiniciar la escena de simulación. De este modo, si el usuario se ha equivocado con las instrucciones que ha programado para el robot, al pulsar el botón de reinicio se detendrá la ejecución del código del *Workspace* y se llevará el robot hasta el punto de inicio (**Figura 52**).

Después de algo de búsqueda por el código fuente de la librería UBlockly, descubrí que existe una función que detiene la ejecución del intérprete de código C# mientras se conservan los bloques que se han puesto en el *Workspace*, por lo que no se pierde el progreso de la construcción del script.

En la **Figura 51** podemos observar la función del nuevo fichero **ResetTransform** que se ejecuta cuando queremos reiniciar la escena.

```
    public void ResetPosRot(){
        robot_tr.position = new Vector3(131.4f,8.5f,-76.1f);
        robot_tr.eulerAngles = new Vector3(0,0,0);

        CSharp.Interpreter.Stop();//Detención del intérprete
        //Deteniendo robot
        robot_rb.constraints = RigidbodyConstraints.FreezeRotation |
        RigidbodyConstraints.FreezePositionX |
         RigidbodyConstraints.FreezePositionZ;

        wheel_r.useMotor = false; wheel_r_rb.drag = 9999999;
        wheel_l.useMotor = false; wheel_l_rb.drag = 9999999;
        wheel_c.useMotor = false; wheel_c_rb.drag = 9999999;
    }
```

Figura 51: Función de reinicio de la simulación

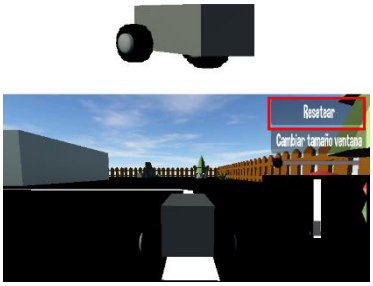


Figura 52: Botón de reinicio de simulación

Además de detener la interpretación del código y devolver el robot a la posición de origen, también se apagan los motores de las ruedas y se aumenta la fricción de estas con el suelo, junto con restricciones de movimiento salvo en el eje Y. Esto provoca que cualquier movimiento que estuviera haciendo el robot sea completamente detenido.

Vuelta a la escena de creación

Nos planteamos que el usuario por accidente podría equivocarse con el diseño del robot, por lo que se decidió incluir un botón en la escena de simulación que permitiera volver al menú de creación, para que pueda corregir los problemas que tenga su robot y volver al entorno a probar su diseño (**Figura 54**).

En la **Figura 53** podemos ver que para cambiar de escena hemos añadido unas líneas al fichero **MainMenu.cs** que detienen la interpretación de código C# en el caso de que se estuviera ejecutando algún bloque en la simulación. Este método se ejecuta al pulsar el botón "*Volver a creación".*

```
public void CreateRobotSq () {
    CSharp.Interpreter.Stop();
    SceneManager.LoadScene ("CreationSq");
}
```

Figura 53: Regreso a la escena de creación y parada del intérprete

Figura 54: Botón de retorno a la escena de creación

```
void Start () {
    ...
    parentFront = GameObject.Find("Permanente/Robot/FrontSide").GetComponent<Transform>();
    placeFront = GameObject.Find("Permanente/Robot/FrontSide/FrontTransform").GetComponent<Transform>();
    placeFrontIR = GameObject.Find("Permanente/Robot/FrontSide/FrontTransformIR").GetComponent<Transform>();
    ...
}
```

Figura 55: Fragmento de código de asignación de lados del robot al fichero CrearSensor.cs

Existieron algunos problemas a la hora de implementar esta funcionalidad, ya que los scripts para realizar la personalización de los sensores del robot no almacenaban las referencias a los GameObjects de los distintos lados del robot. Esto es debido a que eran referenciados por medio del inspector del editor de Unity, provocando que al volver a la escena de creación no se pudiera editar el robot. La solución consistió en hacer la referencia directamente en el código en la función "*Start*" del fichero **CrearSensor.cs**. Se puede ver un fragmento de este código en la (**Figura 54**)**,** en el que asignamos los GameObjects de la parte delantera a sus variables en el código. Debe hacerse lo mismo para el resto de las partes del robot (*Back, Left, Right y Top*)**.**

Esta complicación también ocurría al intentar cambiar el tamaño de las ruedas y del chasis del robot, aunque en este punto se planteó que esta funcionalidad no tenía tanta importancia como se había planteado el pasado curso académico, por lo que decidimos eliminar los botones que permitían editar estos aspectos del modo de creación, contando únicamente con la funcionalidad de editar sensores.

Aparte de esto, también se tuvieron que definir algunos parámetros en las funciones "*OnEnable"* de los ficheros **Rotation** y **RobotCollider** para cambiar los parámetros de robot no solo una vez, (que es lo que nos permite la función "*Start*") sino cada vez que se pasa de una escena a otra y se habilitan estos scripts. Entre estos cambios destacan el de la cámara "Robot Camera", que debe volver a ocupar la totalidad de la pantalla al cambiar a la escena de creación.

Ocultar bloques de sensores no presentes en el diseño del robot

Para evitar fallos y confusiones a los usuarios de la aplicación a la hora de crear los scripts con los bloques, decidimos ocultar aquellos relacionados con sensores que no se han añadido a su robot, ya que no necesitará usarlos.

Examinando detenidamente la librería UBlockly encontramos la función en la que se instancian los bloques en las distintas categorías. Hicimos unos cambios en esta, de modo que se comprueba el bloque que se pretende instanciar. Si este bloque es de la categoría “SENSOR”, se comprueba que el robot cuenta con algún sensor que esté relacionado con el bloque que estamos evaluando, y solo en el caso de que lo encuentre, incluye el bloque en la vista de la categoría (**Figura 56**).

```
protected virtual void BuildBlockViewsForActiveCategory(){
            Transform contentTrans = mRootList[mActiveCategory].transform;
            var blockTypes = mConfig.GetBlockCategory(mActiveCategory).BlockList;
            foreach (string blockType in blockTypes){
                switch(blockType){ //Mostrar solo bloques de los sensores con los que cuenta el robot
                    case "sensor_ir_white":
                        if (GameObject.FindObjectOfType<IRSensorDetection>() != null){ //Si existe algún sensor
                            NewBlockView(blockType, contentTrans); //Añade el bloque a la vista de la categoría
                        }
                        break;
                    //Comprobaciones para el resto de los bloques y sensores
                    default:
                        NewBlockView(blockType, contentTrans);
                        break;
}}}
```

Figura 56: Modificación para ocultar bloques de sensores (ClassicToolbox.cs)

En la **Figura 57** podemos ver el resultado que hemos obtenido al realizar esas modificaciones al código de instanciación de bloques de nuestra *toolbox*.

Figura 57: Categoría SENSOR con los bloques relacionados con los sensores del robot

Aumentar y disminuir el tamaño de los bloques de área de trabajo

A pesar de poder arrastrar el cursor para poder ver la totalidad de nuestro script visual, hemos notado que el área de trabajo podría ser algo incómoda en el caso de que tengamos muchos bloques en ella. Por eso decidimos incluir unos botones en la esquina inferior izquierda por medio de los cuales podremos aumentar y disminuir el tamaño de los bloques que se encuentran en el *Workspace* (**Figura 59**).

Se escribió un pequeño script (**ModCodingPanelSize.cs**) con dos funciones, que son llamadas al pulsar los botones para aumentar y disminuir la escala del *Workspace*. Podemos ver estas funciones en la **Figura 58.**

```
public void MakeitBigger(){
    if(codingpaneltr.localScale.x < 1.33){
        codingpaneltr.localScale += new Vector3(0.11f,0.11f,0);
}}
public void MakeitSmaller(){
    if(codingpaneltr.localScale.x > 0.66){
        codingpaneltr.localScale -= new Vector3(0.11f,0.11f,0);
}}
```

Figura 58: Cambiar escala del panel de codificación

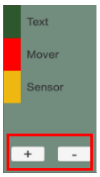


Figura 59: Botones para cambiar escala del panel de codificación

3.3.4 Creación de retos

Una vez se llevaron a cabo la integración de todos los módulos en uno solo y se realizaron mejoras en la funcionalidad e interfaz de la aplicación, llegamos a la fase de desarrollo en la que creamos los retos. Estos retos pondrán a prueba las habilidades de los usuarios y la capacidad del robot de solucionar situaciones por medio de las instrucciones que le proporcionemos y el uso de sus sensores.

Se diseñaron un total de dos retos para el robot. Uno de ellos para el sensor de infrarrojo, que consiste en seguir dentro de una línea blanca para llegar a su objetivo. El otro sería para los sensores de contacto y ultrasonido, en el que el robot tendrá que recorrer un tramo laberíntico hasta capturar una moneda.

Cada uno de estos retos es una escena distinta, ya que el escenario ha sido modificado para cada una de estas pruebas, aunque el resto de los elementos son exactamente iguales. Para poder resolver las pruebas, los usuarios deberán tener información sobre cómo es el terreno del entorno de simulación.Se han preparado dos botones en la escena de simulación que nos permiten ir a cada uno de estos retos.

Reto del sensor infrarrojo

El sensor infrarrojo permite evaluar si una zona del terreno por la que está pasando el robot es blanca o es negra. Basándonos en las capacidades de este sensor, se ideó una prueba que consistiría en un camino blanco sobre un fondo negro que tiene al final una moneda. Para recoger la moneda, se deberá aprovechar la información que reciba el sensor con el fin seguir la ruta y superar el reto.

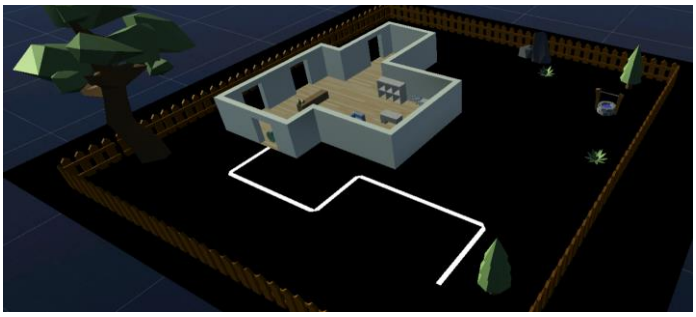


Figura 60: Entorno de simulación: Reto para Sensor Infrarrojo

Como podemos ver en la **Figura 60,** el reto cuenta con un camino que tiene tanto rectas como curvas, por lo que el usuario deberá hacer buen uso de los sensores para superar esta prueba.

Para crear el reto se tuvo que añadir un “Material” de color negro al GameObject “Plane” para pintar el plano de este color. El camino fue creado a base de otros pequeños GameObjects del mismo tipo que “Plane” de color blanco, a los que se le modificaron la escala, posición y rotación para unirlos y formar un camino. Las curvas fueron hechas simplemente creando otro objecto “Plane” de color negro y superponiéndolo en las esquinas como se puede apreciar en la **Figura 61**.

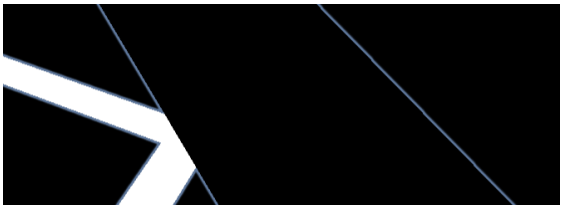


Figura 61: Plane negro superpuesto

Una posible solución a este reto consistiría, en primer lugar, en la creación de un robot que cuente con dos sensores infrarrojos con precisión máxima y alcance rondando los 0.3 metros. Los sensores deben estar situados en la parte delantera, posicionados uno en cada extremo lo más cercanos al suelo posible, tal y como se muestra en la **Figura 62**.

Figura 62: Configuración de robot: Reto para Sensor Infrarrojo

Después, se debería elaborar a base de la unión de distintos bloques el script de la **Figura 63.**

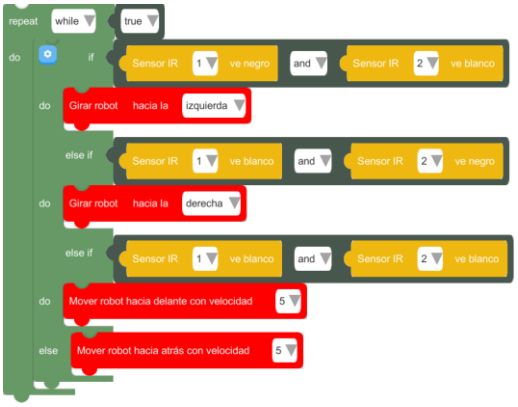


Figura 63: Script para superar prueba: Reto para Sensor Infrarrojo

Como podemos ver a simple vista, es un script bastante sencillo, aunque usamos algunos elementos bastante interesantes, como son los bucles y los condicionales. Básicamente el robot podrá hacer una de las cuatro acciones que podemos ver en la lista del siguiente párrafo, que estarán condicionadas por lo que detecten los sensores infrarrojos.

1. Si ambos sensores reciben “blanco”, el robot **avanza**.
2. Si el sensor situado a la derecha recibe “negro” y el izquierdo detecta “blanco”, el robot realizará un **giro a la izquierda**.
3. Si el sensor situado a la izquierda recibe “negro” y el derecho detecta “blanco”, el robot realizará un **giro a la derecha**.
4. Si ambos sensores reciben “negro”, el robot **retrocede**.

Reto del sensor de contacto y sensor ultrasonido

El sensor de contacto envía una señal en el caso de que detecte una colisión con otro objeto. Mientras, el sensor ultrasonido capta la distancia a la que se encuentra un objeto en un rango de 2.5 metros. Con esto en mente, barajamos la posibilidad de hacer un reto que pudiera solucionarse usando alguno de estos dos sensores, ya que, si lo pensamos bien, las capacidades de ambos son relativamente similares por el hecho de que los dos detectan la ubicación de un cuerpo usando técnicas distintas. Se ideó una prueba que permite explotar las posibilidades de ambos sensores, en

el que el objetivo será recorrer un camino laberíntico hasta el final para recoger una moneda. El laberinto consiste en la unión de varios rectángulos alargados, que son inamovibles.

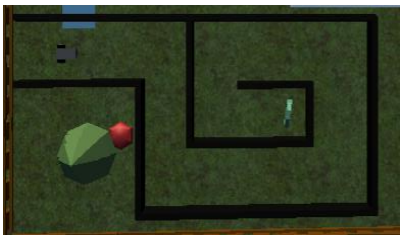


Figura 64: Entorno de simulación (Vista aérea): Reto para Sensor de Contacto y Ultrasonido

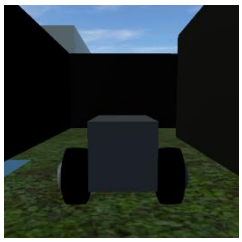


Figura 65: Entorno de simulación (Vista robot): Reto para Sensor de Contacto y Ultrasonido

Tal y como vemos en la **Figuras 64** y **65,** el camino cuenta con varias curvas y paredes que dificultarán recorrer el camino. Para llegar hasta el final del recorrido, podemos crear algunos de los robots que se mostrarán en las **Figuras 66** y **67.**

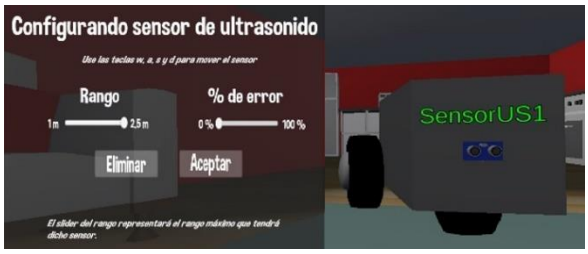


Figura 66: Configuración de robot: Reto para Sensor Ultrasonido

Figura 67: Configuración de robot: Reto para Sensor de contacto

Ambos sensores deben estar colocados en el frontal del robot, de modo que detecten los objetos por delante. El rango del sensor ultrasonido lo hemos establecido a 2.5 metros para que tenga bastante anticipación a la hora de detectar las paredes del laberinto.

En las **Figuras 68** y **69** tenemos unos ejemplos de scripts que conseguirían hacer que el robot recorra el camino de forma satisfactoria.

Figura 68: Script para superar prueba: Reto para Sensor de contacto

Figura 69: Script para superar prueba: Reto para Sensor Ultrasonido

Estos scripts son bastante similares en cuanto a las condiciones con las que cuentan, ya que, para ambos casos, hay que realizar el mismo recorrido. Este recorrido sería girar a la derecha 90 grados una vez y girar a la izquierda 90º cinco veces. Hay algunas variaciones, como, por ejemplo, el sensor de contacto retrocede bastante al chocar con la pared para ganar espacio a la hora de realizar el giro. Por otra parte, el script de sensor ultrasonido apenas retrocede al detectar el muro, ya que el robot estará lo suficientemente alejado de la pared como para poder girar correctamente.

Otra de las diferencias que existen son las condiciones para ejecutar esas acciones. Mientras que el robot con el sensor de contacto debe avanzar si no hay choque, el robot del sensor ultrasonido avanza hasta que la distancia entre él y la pared sea menor que tres metros.

La curiosidad de este reto es que a pesar de ser el mismo, debemos pensar de forma distinta al afrontarlo con un sensor u otro, ya que la dificultad radica en entender cómo se comportan y comprender que devuelven distinta información.

3.3.5 Exportar proyecto a HTML5

Después de haber finalizado el desarrollo del proyecto, el último paso era compilarlo para poder implementarlo como aplicación web. Esta tarea es bastante sencilla, ya que simplemente nos hemos tenido que dirigir a la barra de herramientas del editor de Unity y pulsar en *File->Build Settings* (**Figura 70)**. Al hacer esto, vemos que se abre una ventana que nos muestra varias plataformas a la que exportar, entre las que debemos seleccionar HTML5, para a continuación, pulsar *Build And Run* (**Figura 71**). Después de pulsar este botón, debemos elegir la carpeta destino para almacenar la exportación del proyecto.

Una vez pasados unos quince minutos ya estará exportado. Nos dirigiremos a la carpeta donde se han guardado estos ficheros y arrastramos el archivo **index.html** al navegador (en mi caso he usado Mozilla Firefox), donde podemos comprobar que se lanza correctamente (**Figura 72**).

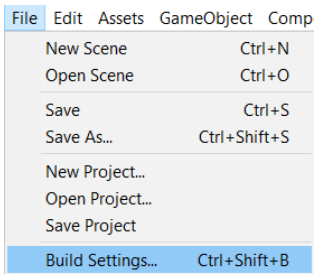


Figura 70: File->Build Settings en el editor de Unity

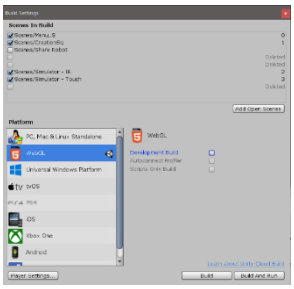


Figura 71: Ventana de exportación de proyecto

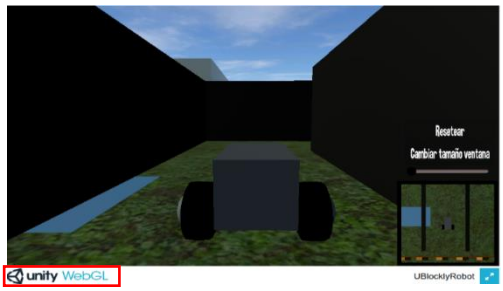


Figura 72: Proyecto en el navegador

Capítulo 4
Caso de uso

Al iniciar la aplicación lo primero que veremos será este pequeño menú (**Figura 73**).

Figura 73: Menú de inicio

Debemos pulsar sobre la opción “Crear un robot” para pasar a la escena de creación. Una vez aquí, debemos personalizar nuestro robot para poder superar alguno de los dos retos propuestos. En este caso, vamos a orientar este caso de uso a la resolución del reto de sensor de ultrasonido.

Si queremos añadir el sensor de ultrasonido, debemos dirigirnos a la parte izquierda, en la que tenemos las opciones para añadir sensores y pulsar sobre el botón “Ultrasonido”. Una vez hayamos pulsado este botón, se añadirán algunas opciones a la pantalla, que nos permitirán elegir la parte del robot en la cual colocaremos el sensor. En este caso, vamos a pulsar sobre la opción “Parte frontal” (**Figura 74**).

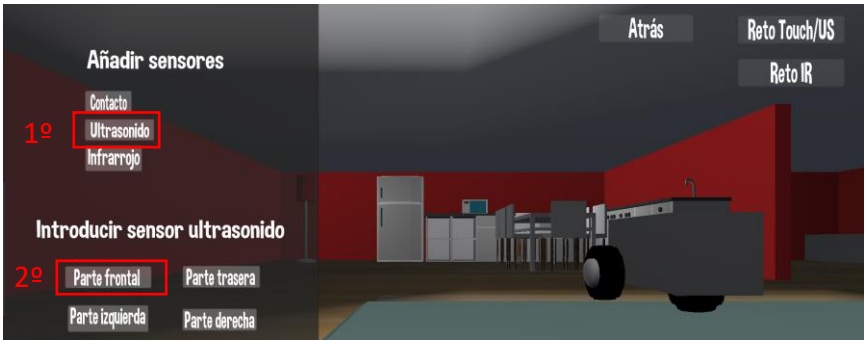


Figura 74: Añadiendo un sensor de ultrasonido a la parte delantera del robot.

Cuando pulsemos ese botón se nos desplegará un menú con distintas opciones para personalizar de este sensor, en este caso, podemos alterar el rango de detección de objetos del sensor y la precisión de aciertos que va a tener. Si arrastramos las barras situadas a la izquierda, podremos alterar estos parámetros. Para cambiar la posición del sensor pulsaremos las teclas W,A,S y D. Nosotros vamos a situar el sensor aproximadamente en el centro de la parte frontal con la configuración de precisión y rango que vemos en la **Figura 75.** Cuando hayamos terminado de configurar este sensor, pulsamos en el botón “Aceptar”.

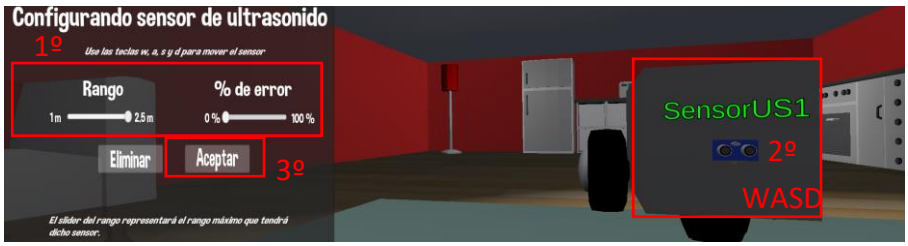


Figura 75: Personalizando sensor ultrasonido

De vuelta en el menú de creación, pulsaremos en el botón "Reto Touch/US" para ir al entorno de simulación con la prueba correspondiente. En el caso de que hayamos cometido algún error en la configuración del sensor ultrasonido, se podrá tanto modificar como eliminar si pulsamos con nuestro cursor encima de él (**Figura 76**).

Figura 76: Preparativos para iniciar el reto de sensor ultrasonido

Al pulsar el botón del reto, se cargará la escena de simulación junto con el área de trabajo sobre la que crearemos nuestro script visual. Para comenzar a programar, debemos pulsar sobre alguna de las categorías de la *toolbox*, momento en el que se desplegarán los bloques de esa categoría. Acto seguido elegiremos alguno de esos bloques y los arrastraremos hasta el área de trabajo (**Figura 77**).

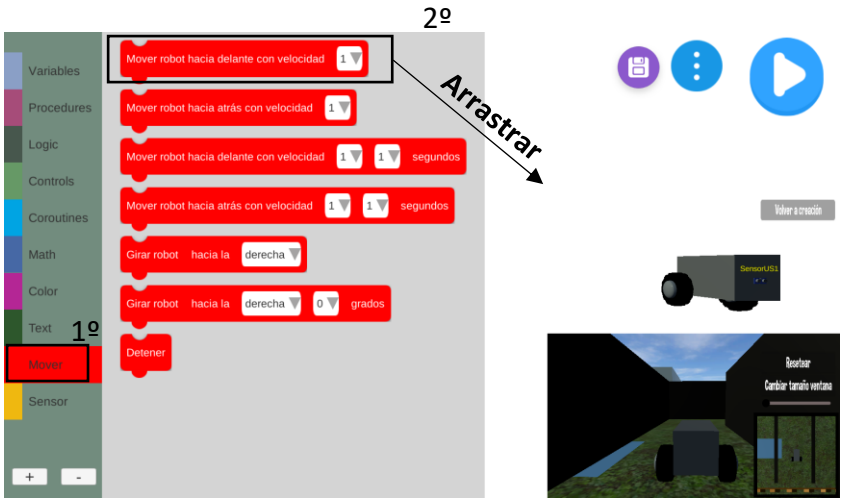


Figura 77: Llevar un bloque al área de trabajo

Una vez tengamos este bloque en el *Workspace*, vamos a tener que encadenar distintos bloques para formar el script de la **Figura 69.** Encadenar bloques es tan sencillo como arrastrar un bloque al área de trabajo y acercarlo al que queremos unirlo. Cuando estén lo suficiente cerca, la ranura por la que se van a encajar los bloques cambiará de color. Si en este momento soltamos el bloque que estamos arrastrando, quedarán unidos (**Figura 78**).

Figura 78: Encajar dos bloques

En algunos bloques vamos a tener campos de selección o de introducción numérica. Cuando pulsemos sobre estos nos saldrá un cuadro de diálogo. Con estos podemos especificar la velocidad del robot, los ángulos de giro, la dirección, o el sensor del que recibimos la información (**Figuras 79** y **80**).

Figura 79: Cuadro de diálogo: Introducción de número

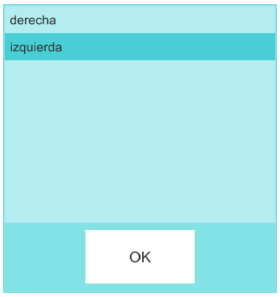


Figura 80: Cuadro de diálogo: Selección de una opción

Después de encadenar bloques conseguiremos el script que resolverá el reto. Para ejecutarlo debemos pulsar en el botón situado en la parte superior derecha. Una vez haya dado comienzo la simulación, podemos observar tanto la traza de ejecución del código por medio de un punto verde, (que se irá posicionando encima del bloque que está ejecutando el intérprete), como el avance del robot por el escenario. Si en algún punto de la ejecución queremos reiniciar la escena porque algo ha ido mal, debemos pulsar sobre el botón “Resetear” para volver al punto de partida (**Figura 80**).

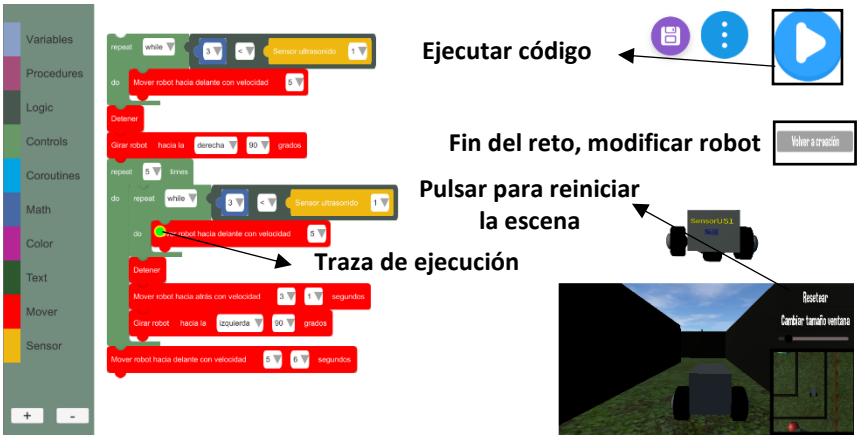


Figura 81: Ejecutando bloques del área de trabajo

Si conseguimos que la ejecución haya sido satisfactoria y queremos volver a la escena de creación, solo debemos pulsar sobre el botón “Volver a creación” y cambiar el diseño del robot para resolver otro reto.

Capítulo 5
Conclusiones

A modo de conclusión, podemos dar por cumplidos los objetivos que se habían planteado para este trabajo de fin de grado, ya que no solo se han conseguido integrar de forma más sólida los tres módulos, sino que se han unido usando únicamente la tecnología Unity.

El hecho de que todos los módulos estén condensados dentro de una misma aplicación permite que su uso sea más fácil y cómodo para el usuario, ya que, si lo comparamos con la versión anterior de esta línea de trabajo, debíamos cargar un total de tres veces ficheros de configuración para aplicaciones distintas. Ahora todo está localizado en una única aplicación, que se ha conseguido hacer compatible con navegadores, lo que permitirá que cualquiera pueda usarla fácilmente al alojarla en un servidor web.

Además de la mejora de usabilidad, se han añadido muchas mejoras tanto a nivel de funcionalidad como a nivel visual. Por ejemplo, poder ampliar la pantalla del simulador, cambiar el tamaño de los bloques o poder reiniciar la simulación son valores añadidos que enriquecen la experiencia de uso.

No nos debemos olvidar de que también se han añadido algunos retos, ya que, sin estos, la aplicación queda algo vacía ya que no podemos poner a prueba los robots que creamos. Es cierto que tan solo hay dos retos, pero son suficientes para hacer una demostración de que todos los módulos han sido debidamente integrados y que todo funciona.

Este proyecto sienta las bases de una aplicación de robótica educacional que tiene mucho potencial de crecimiento, por lo que en futuras líneas de trabajo se podrían añadir más retos, además de catalogarlos por orden de dificultad, e incluso crear un sistema por el que los usuarios puedan hacer sus propios retos de forma fácil. Teniendo esto en mente, sería un gran valor añadido que la aplicación formara parte de una comunidad educativa online, de modo que se pudieran subir y compartir los retos que se creen entre todos los usuarios, además de añadir algún sistema que permita puntuar y hacer rankings según la destreza de los usuarios a la hora de resolver las pruebas.

Otras mejoras que se podrían desarrollar en futuros trabajos serían añadir otros tipos de chasis al robot que permitan una distribución distinta de las ruedas, ya que esto podría dar bastante juego a la hora de plantear los retos, o incluso añadir partes móviles, por medio de las cuales el autómata pueda recoger elementos del escenario y transportarlos.

Capítulo 6
Summary and Conclusions

In conclusion, we can say that the objectives for this end-of-degree work has been accomplished, since not only have the three modules been more robustly integrated, as well have been integrated using Unity technology.

The fact that all modules are integrated within the same application makes their use easier and more comfortable for the user, because, if we compared it with the previous version of this line of work, we should load a total of three times configuration files in different applications. Now everything is condensed into a single application that was able to be exported to HTML5, which will allow anyone to use it easily if we host it on a web server.

In addition to the improved usability, many improvements have been added both at the functionality level and at the visual level, for example, being able to expand the simulator screen, resize the blocks or be able to restart the simulation are elements that enrich the experience of use.

We must not forget that some challenges have been added, because without them, the application could look empty for the fact that we couldn´t test the robots that we create. It is true that there are only two challenges, but they are enough to demonstrate that all modules have been properly integrated and that everything it's working.

This project lays the foundation for an educational robotics application that has a lot of growth potential, so in future lines of work more challenges could be added, as well as classify them in order of difficulty, and even create a system by which users can easily make their own challenges. With this in mind, it would be great input if the app were part of an online educational community, so that challenges that are created among all users could be uploaded and shared, as well as adding some system that allows to score and do rankings according to the prowess of users in solving tests.

Other improvements that could be developed in future work would be to add other types of chassis to the robot allowing for a different distribution of wheels, since this could give a lot of possibilities when it comes to setting the challenges, or even adding moving parts, through which the automaton can pick up elements from the stage and transport them.

Capítulo 7
Presupuesto

En la **Tabla 2** podemos ver el prepuesto correspondiente al trabajo realizado en este proyecto.

7.1 Presupuesto

Tarea	Precio por hora	Horas empleadas	Coste
Investigación acerca de las tecnologías empleadas en el proyecto y familiarización con el mismo	14 €	35h	490€
Integración del módulo de creación y simulación	14 €	40h	560 €
Integración de UBlockly en el proyecto	14 €	70h	980 €
Funcionalidades adicionales	14 €	40h	560 €
Creación de retos	14 €	15h	210 €
Testeo del proyecto	14 €	60h	840 €
Elaboración de la memoria	14 €	50h	700 €
TOTAL		310h	**4340 €**

Tabla 2: Tabla de presupuesto

Bibliografía

[1] Robótica educativa – Wikipedia **(Accedido el 14-02-19)** https://es.wikipedia.org/wiki/Rob%C3%B3tica_educativa

[2] ¿Qué es la robótica educativa y por qué nos gusta tanto? **(Accedido el 14-02-19)** https://www.juguetronica.com/blog/que-es-la-robotica-educativa-y-por-que-nos-gusta-tanto/

[3] Pensamiento computacional – Wikipedia **(Accedido el 15-02-19)** https://es.wikipedia.org/wiki/Pensamiento_computacional

[4] Beneficios del pensamiento computacional **(Accedido el 15-02-19)** https://codelearn.es/beneficios-del-pensamiento-computacional/

[5] Blockly | Google Developers **(Accedido el 20-02-19)** https://developers.google.com/blockly/

[6] Qué es la robótica educativa | Nuevo sistema de enseñanza **(Accedido el 26-02-19)** https://edukative.es/que-es-la-robotica-educativa/

[7] B. Ortega-Ruipérez and M. A. Mikel, "Robótica DIY: pensamiento computacional para mejorar la resolución de problemas," vol. 17, no. 2, pp. 129–143, 2018.

[8] Capítulo 2: Abstracción **(Accedido el 25-02-19)** http://www.pensamientocomputacional.org/Files/02Capitulo.pdf

[9] Coding Robots Curriculum Outline – CoderZ **(Accedido el 02-03-19)** https://gocoderz.com/coding-robots/

[10] Educators – CoderZ **(Accedido el 02-03-19)** https://gocoderz.com/educators/

[11] Vrep: Simulación de robots virtuales – robologs **(Accedido el 02-03-19)** https://robologs.net/2016/01/22/vrep-simulacion-de-robots-virtuales/

[12] Coppelia Robotics V-REP: Create. Compose. Simulate. Any Robot: Features **(Accedido el 02-03-19)** http://www.coppeliarobotics.com/features.html

[13] Robot Dash de Wonder Workshop - Apple **(Accedido el 02-03-19)** https://www.apple.com/es/shop/product/HJYC2VC/A/robot-dash-de-wonder-workshop

[14] Unity (Motor de Juego) – Wikipedia **(Accedido el 05-03-19)** https://es.wikipedia.org/wiki/Unity_(motor_de_juego)

[15] Products – Unity **(Accedido el 05-03-19)** https://unity3d.com/es/unity

[16] ¿Sabes que es UNITY? Descúbrelo aquí – De Idea a App **(Accedido el 14-03-19)** https://deideaaapp.org/sabes-que-es-unity-descubrelo-aqui/

[17] Creando y usando scripts - Unity Manual **(Accedido el 14-03-19)** https://docs.unity3d.com/es/current/Manual/CreatingAndUsingScripts.html

[18] Rigidbody - Unity Manual **(Accedido el 15-03-19)** https://docs.unity3d.com/es/current/Manual/class-Rigidbody.html

[19] Prefabs – Unity Manual **(Accedido el 15-03-19)** https://docs.unity3d.com/es/current/Manual/Prefabs.html

[20] Escenas - Unity Manual **(Accedido el 15-03-19)** https://docs.unity3d.com/es/current/Manual/CreatingScenes.html

[21] GitHub imagicbell/UBlockly: Reimplementation of Google Blockly for Unity

(Accedido el 02-04-19)
https://github.com/imagicbell/UBlockly

[22] Google Blockly Reimplementation With Unity/C#(1) Magicbell's Blog **(Accedido el 02-04-19)**
http://magicbell.beanstu.io/unity/2017/10/11/blockly-one.html

[23] Google Blockly Reimplementation With Unity/C#(2) Magicbell's Blog **(Accedido el 02-04-19)**
http://magicbell.beanstu.io/unity/2017/10/14/blockly-two.html

[24] Google Blockly Reimplementation With Unity/C#(3) Magicbell's Blog **(Accedido el 02-04-19)**
http://magicbell.beanstu.io/unity/2017/10/22/blockly-three.html

[25] Google Blockly Reimplementation With Unity/C#(4) Magicbell's Blog **(Accedido el 02-04-19)**
http://magicbell.beanstu.io/unity/2017/10/31/blockly-four.html

[26] Rodríguez Rivarés, Andrea, "Robótica educativa mediante programación visual", 2018.
https://riull.ull.es/xmlui/handle/915/10306

[27] Paz González, Eduardo de la, "Simulador de robots para fomentar el pensamiento computacional a través de lenguajes de programación visual", 2018.
https://riull.ull.es/xmlui/handle/915/9409

[28] Jurassic: A .NET library to parse and execute JavaScript code. **(Accedido el 20-03-19)**
https://github.com/paulbartrum/jurassic

[29] Repositorio público del proyecto **(Accedido el 28-06-19)**
https://github.com/alu0100891843/UBlocklyRobot

[30] GitHub Pages UBlocklyRobot_web **(Accedido el 02-07-19)**
https://alu0100891843.github.io/UBlocklyRobot_web/